FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Generating Entertaining Platform Game Levels

Nelson André Amaral Oliveira

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Prof. Henrique Lopes Cardoso

Co-Supervisor: MSc. Luís Teófilo

July 14, 2014

Generating Entertaining Platform Game Levels

Nelson André Amaral Oliveira

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Prof. António Augusto de Sousa
External Examiner: Prof. Luís Paulo Gonçalves dos Reis
Supervisor: Prof. Henrique Daniel de Avelar Lopes Cardoso

July 14, 2014

Abstract

Platform games have become less popular with the evolution of the video game industry. Nevertheless, with the recent growth in popularity of mobile devices, they become yet again popular, due to their simplicity in mechanics and game play. Since their popularity is increased, they can be made even simpler, far more addicting and entertaining by utilizing "machine learning" techniques, consequently reducing development costs. This is where *procedural content generation* comes in play, since creating levels manually is time and money-consuming. These techniques will allow games to have their longevity increased, and become far more entertaining.

This dissertation consisted in the usage of machine learning techniques to generate platform levels for the known platform game *Super Mario Bros.*, having the *fun* factor severely emphasized. For that to happen, several *procedural content generation* techniques have been utilized, from the simplest *level portion* to the complex *genetic algorithm* based *tournament selection*. These levels are also parametrised by each user's likings, throughout certain features such as enemy density, the selection of far more common platforms or the difficulty itself, which is based on each player's real performance.

Two *social experiments* have been performed where majority (90%) of participants considered these levels similar to manually designed ones, augmenting the credibility of said generated levels. These experiments were conducted with players as participants, therefore also boosting the credibility of the content generator.

Resumo

Os jogos de plataformas tornaram-se cada vez menos populares com a evolução da indústria dos vídeo-jogos. No entanto, com o aumento da popularidade dos dispositivos móveis, estes emergem novamente devido à sua simplicidade, tanto em mecânicas de jogo como jogabilidade. Visto que a sua popularidade está em vias de aumentar, estes conseguem ser desenvolvidos de forma mais simples, mais viciante e entretida ao serem utilizadas técnicas de aprendizagem, reduzindo os custos de desenvolvimento. É aqui que a *geração procedimental de conteúdos* é inserida, visto que criar níveis de forma manual é dispendioso em termos de tempo e dinheiro. Estas técnicas irão então permitir que a longevidade dos jogos aumente, tornando-se consequentemente mais divertidos.

Esta dissertação consiste na utilização de técnicas de *machine learning* para gerar níveis de plataformas para o conhecido *Super Mario Bros.*, tendo sempre em vista a componente do *divertimento*. Para esse efeito, diversas técnicas de *geração procedimental de conteúdos* foram utilizadas, desde a simples *porção do nível* até aos complexos *algoritmos genéticos* baseados em *selecção por torneios*. Estes níveis são também parametrisados segundo as preferências dos utilizadores, através de funcionalidades como a densidade de inimigos, a seleção de plataformas mais comuns ou até a dificuldade, decidida pela *performance* real do jogador.

Duas *experiências sociais* foram realizadas, onde a maior parte dos participantes (90%) consideraram todos os níveis automáticamente gerados bastante semelhantes aos níveis desenhados manualmente, o que aumenta a credibilidade do gerador. Visto que estas experiências tiveram jogadores como participantes, a credibilidade do gerador de conteúdos é também elevada a um nível maior.

Acknowledgements

Upon completion of this dissertation project, I'd love to thank all of the professors that accompanied me in this academic journey, especially my co-supervisors. Without their support, this project would have never been possible. However, an academic course is not simply done with teachers and theory. My friends have given me everything I could have ever asked for, and for that, I am truly grateful. To all of my academic partners, and friends for life, I express my gratitude.

Additionally, I'd like to leave my acknowledgements to my family. Their support has been magnificent, and I know I will always be able to count with them for all of my future ventures. I can't express how deeply thankful I am for being part of such great family.

Finally, I cannot restrain from mentioning my girlfriend, who has put up with me throughout the whole project. A special thank you to her, for being there for me.

Nelson Oliveira

"It's not a question of can or can't.
There are some things in life you just do."

Lightning - *Final Fantasy XIII*^{TM}

Contents

List of Figures

List of Tables

Abbreviations

PCG	Procedural Content Generation
SMB	Super Mario Bros.
IMB	Infinite Mario Bros.
AI	Artificial Intelligence
IEEE	*Institute of Electrical and Electronics Engineers*
NES	Nintendo Entertainment System

Chapter 1

Introduction

The provided dissertation aims to develop a generator that will create entertaining levels, the research developed in order to achieve such goal, as well as the proper project verification and validation. Throughout the whole document, several examples of games, levels, level zones and algorithms are explained to detail. This first chapter describes the dissertation's purpose, as well as delimiting its chapters. Additionally, the general dissertation's hypothesis will also be explicitly described, being later on enhanced by results and conclusions.

1.1 Context and Motivation

The following dissertation comes within the context of *platform games*, and their ability to be manipulated and therefore having their content generated automatically. In order to do so, several generation techniques will be considered in the fields of *procedural content generation*, as well as existing projects in these fields. *Platform games* have become viral ever since the 1980s (at the end of the decade, *Super Mario 3* sold over 11 million copies [Lai04]), but lost their popularity with the evolution of the video game industry, considering the areas of computer graphics and realism. However, with the emergence of mobile devices, these games' simplicity was again considered and easily ported into the devices, regaining popularity. The majority of the content on these games is created and designed manually, though. In order to overcome the drawbacks associated with this manual work, this dissertation will explore how *procedural content generation* can be used to reduce development efforts and costs.

1.2 Objectives and Hypothesis

The objective of the dissertation is to enable the creation of levels supported by procedural generation tools, as previously stated. However, these levels ought to be *entertaining*[1] to the people

[1] Whenever a player feels joy while playing a game.

playing them. Additionally, this dissertation will explore whether it is possible to create automatic levels that will be deemed as manually created ones. In resume:

- *Fabricate a procedural level generator* - Create a parametrisable level generator according to the studied algorithms;
- *Unique and immerse game mechanics* - Innovate in the way platform games are played, and introduce interesting mechanics to the basic and existing ones;
- *Nested artificial intelligence* - Associate artificial intelligence to the *fun* factor of the generated levels;
- *Validate generated levels* - Conduct experiments to validate whether the generated levels can be compared to manual ones and whether they are actually *fun*.

In order to complete the above, social experiments have been conducted to discover which would be the best content to place in generated levels. Furthermore, several programming and game designing techniques have been taken into account, with the sole purpose of creating a level generator that will provide an immerse and entertaining experience to its players. Each generated level uses rules and parameters that will increase its credibility.

1.3 Developed work summary

Throughout the dissertation project, an entertaining level generator has been developed. This generator allows for its users to select parameters according to their own preferences, through the first panel presented. These parameters include the selection of level types, kinds, difficulty, length, height, time limit and respective generation probabilities and fields. As soon as these parameters have been chosen, the user will then be transported to the game scenario where he/she can play *Mario* within the level generated by the program. The majority of the elements in the screen are placed according to the probability parameters previously set. The user is able to control *Mario*, and travel throughout the generated environment. Additionally, users can store their levels, so that these can be loaded through the first panel presented any time later.

1.4 Document's structure

Apart from the current chapter, this document attains 6 more chapters.

Chapter 2 presents an insight towards all of the definitions needed to understand the document, as well as extra information needed for the project's development. Concurrently, Chapter 3 deepens into the related work within the area of *procedural content generation*, where related projects and state of the art is presented. The next chapter, being Chapter 4 presents the level generator platform that has been developed, and how it works. Proceeding, Chapter 5 refers to all of the implementation details of the project and the level generator platform. Chapter 6 presents all social

experiments performed, for the project's validation, as well as the gathering of important information from the players within the experiment. Finally, Chapter 7 presents all of the conclusions of this dissertation.

Chapter 2

Background

In order to generate game levels, it is important to understand the history that lies behind platform games, as well as their concept, purpose and target audience. The objective, as previously stated, is to generate levels that take the *fun*[2] factor into account, as well as the immerse experience that the player will be submitted to while considering every single detail that revolves around the term *fun*. In this case, game mechanics, game environments and other aspects need to be considered.

2.1 Platform Games

Platform games (or *platformers*) are games based on areas where the player can jump to, or from (platforms). Usually, there is a hero that plays the role of the main character, and has the task of reaching the end of every single level while avoiding obstacles on its path, throughout platforms, moving over, alongside or even on them, but never falling off of them. Various enemies will attempt to stop the main character from reaching its goal, or even the terrain features the levels[3] will have to offer. Each game has its own mechanics, specific features and other elements that make each one of them unique and therefore different from one another. Notable examples of platformers are the most successful blockbuster franchises of *Super Mario Bros.* and *Sonic the Hedgehog*, which took over the platform game industry rapidly and in an innovative way due to their differences. [LB09]

Whenever a platform game is described, several typical aspects come to mind, such as the controls, their layouts, or how they're organized. Platform games first came by to exist in two dimensions (being these the ones regarded throughout the project), composed by two different axis denominated *X* and *Y*. [SYT13] This implies then that the character can move into four different directions: *up*, *down*, *left* and *right* (as shown in Figure 2.1). Normally, this is the mechanism that allows the player to move their characters in a platformer: pressing the corresponding buttons

[2] Regarding games, fun is achieved with a balance between challenge and entertainment.
[3] Space available to player to complete a certain objective within a video game.

Figure 2.1: 2D reality and axis of a platformer.

on the game pad or keyboard. It is important not to forget the jumping action, since the character will have to go from platform to platform, since the player's freedom in the *Y* axis is not always guaranteed. This is typically associated to an action button that is included in the game pad or is triggered through a predefined key on the player's keyboard.

Another great aspect on platform games is the time meter the player is submitted to. In the majority of them, the player is forced to complete the current level under a certain amount of time, otherwise they will end up losing a *life*. A *life* is a significant counter of how many tries a player has to accomplish the game or a level. If the player fails to complete the level under the specified time, one of the tries is taken away. There is however, a common mechanism that is also present, called *score*. The *score* determines how well the player is doing throughout the game, and how many obstacles or levels the player has surpassed, or even how many collectible items have been found. Normally, it is calculated considering with all of these factors and, henceforth, enhanced every time the player achieves something within the game. There are also several competitive aspects about platformers, since these games have the capability of being easily replayed. With this, players tend to try to achieve maximum performance while completing levels.

2.2 Super Mario Bros.

One of the most awarded and known platformers of all time, *Super Mario Bros.* (1985) tells the tale of a young italian plumber named *Mario* that finds himself with the mission of rescuing *Princess Peach* from the evil claws of *Bowser*, the *Koopa Dragon*. To complete this task, *Mario*

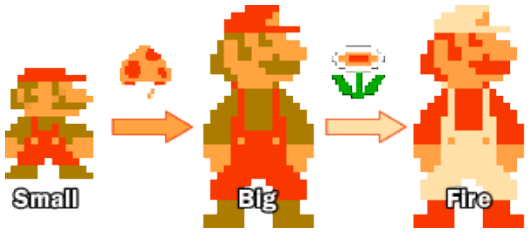

Figure 2.2: Mario's states during *Super Mario Bros.*.

must travel through all of the antagonist's castles in the eight realms of the *Mushroom Kingdom* and search for the princess thoroughly in all of them. However, games in the 1980s were mainly appreciated due to their innovative gameplay and ability to entertain their users. With this, the extended gameplay allied with the platforming experience, make *Super Mario Bros.* one of the most iconic platformers of all time. [LB09]

Super Mario Bros. (SMB) is a simple game, despite all of its complexity. *Mario* has the mission of going through four levels in each of the eight realms. During his journey, obstacles and monsters will try to prevent his goal, so in order to overcome all of the perils, there is an action button that allows Mario to perform a jump. But this is not the only mechanic the game has: there are also *dungeons* and *power-ups*. *Dungeons* are accessible by pipes found in the levels, as external access points. Each *dungeon* contains special traits such as extra collectibles. *Power-ups*[4] in SMB are achieved by collecting three items, which include the *red mushroom*, the *fire flower* and the *star*, and submit *Mario* through several state changes, such as *Small*, *Big*, *Fire* and *Invincible* (see figure 2 - *Invincible* state is not represented since its changes on the main *sprite* are accomplished through *colour variation*). These *power-ups* are obtained in blocks that can be found throughout the levels, marked with a *"?"*. There are also normal blocks that might contain hidden items. [TKK09]

Each of *Mario*'s states determines the abilities he can use, as shown in Figure 2.2. During his normal state (*Small*), *Mario* is completely harmless unless he jumps on enemies that will not have spikes on top of them, and can open up normal blocks to understand what's in them. He then alternates to the *Big* state by acquiring a *red mushroom*. In this state, *Mario* can crush normal blocks that contain no items, by jumping under them. Alas, in *Fire* state, achieved by acquiring a *fire flower*, which is available if and only if *Mario* is in the *Big* state, *Mario* can shoot fireballs at his enemies in order to annihilate them. However, the *Invincible* state is exclusive to all the previous ones, and can be achieved whilst on any other state. With a *star*, *Mario* is completely invincible and can touch any enemy, but will die if he falls into a pit. Regression might also affect the states (i.e.: *Mario* reverting from *Big* to *Small*), in case the character gets hit by an enemy or obstacle that causes damage (except while in the Invincible state). In case the protagonist gets hit while on the *Small* state, the player will lose a try.

This project used the *Super Mario Bros.* engine as a test bed for generating entertaining platform games. With this said, most of the game's mechanics will be included, as well as newly

[4] A special item found in games that will enhance the character's skills.

Figure 2.3: Level generated through PCG techniques.

added ones (refer to Chapter 5.

2.3 Procedural Content Generation

There are techniques to generate levels dynamically, therefore avoiding manual procedures and occupying space in the game's internal memory and potentially reducing development costs. At the dawn of video games, memory availability was a problem, since large quantities were not be actually available to store all of the needed data. With that, the necessity of creating the levels as the player immerses in the game became viral, but only to flunk into oblivion as games took a turn in terms of memory capacity. [AK12] [TKSY11]

Procedural content generation became one of the answers. Basing itself in calculations, and the ability to put together content that will generate a logical level, it is a technique that has been used ever since the 1980s (an example of a level generated with these techniques is shown in Figure 2.3). In its definition, it is the capability of a generator to create content in a procedural way:

> *"Procedural content generation is possible because many things around us, both natural and man-made, have symmetrical properties that can be described mathematically."* (p. 4) [Gu06]

And since content can be generated through mathematics, so can fractals[5]:

> *"Fractals and computers are a marriage made in heaven. One of the most powerful techniques in programming is recursion, whereby a procedure is broken down into a*

[5] A piece of information pertaining a big block of data (more commonly known as *fragment*.

> *sequence of repetitions of itself. A very simple example of this is a brick wall. The procedure build a brick wall is defined in the terms of itself, mainly laying one course of bricks, then build a brick wall on top of it. In practice you must also specify when the procedure stops. In our case a logical limitation to the brick wall would be to stop when it is high enough."* [PJS04]

2.4 Chapter Summary

In this chapter, several definitions were presented so the content presented afterwards can be understood clearly. From platform games (or platformers), to their axis and usual controls, the definition of *procedural content generation* was also emphasized, as well as the description of the *Super Mario Bros.* game for the *Nintendo Entertainment System*. It was also stated that most of the dissertation's project will be based in *Super Mario Bros.*'s game engine, allowing the generator to create levels within the *Mario* universe.

Chapter 3

Related Work

Acquiring relevant knowledge on the existing projects is also of great importance, and allows us to understand a bit more what can be done in order to be innovative. The following sub-sections will tell us about what the industry of video games and the scientific community has accomplished in terms of procedural generation as well as in the field of artificial intelligence, all applied to, of course, platform games.

3.1 Dawn of PCG

When computer games first had their appearance, computers had several limitations in terms of memory and the space that they could allocate to make the games actually work. Consequently, instead of creating the levels one by one and store them in the game's allocated memory, some games would have to rely on the technique of generating procedural content, in order to maintain playability and to enhance the game's ideals. Majority of the games in the decade of 1980 accomplished this technique, but they weren't the only ones. Several games up to today's date have also utilized procedural content generation, and are considered pioneers in the PCG's bundle, such as *Elite*, *Rogue* and *Dwarf Fortress*.

3.1.1 Elite

One of the first games to cause great controversy with memory problems, *Elite* became revolutionary due to its procedural content generation technique and 3D models in the year of 1984. A game dedicated to the exploration of the interstellar universe, resource gathering and the constant adrenaline caused by the obstacles the players encounter, *Elite* managed to surpass all the expectations, as it was written and developed by a pair of undergraduate students, from Jesus College in Cambridge, named David Braben and Ian Bell. An screen shot of the game is presented in Figure 3.1

Figure 3.1: Screen shot of the game *Elite*.

During development, the greatly limited memory became problematic, as they were *fourteen kilobytes*. Back in the time, it was not possible to achieve greater capacities in terms of memory, so the whole source code was suppressed as much as it could be suppressed. Writing the game in Assembly also helped this suppression, allowing the developers to interact directly with the memory itself. However, the original idea was to include 2^{48} (around 282 trillion) galaxies, and generate them all procedurally as the player entered any of them! [Bra06] But this was soon scrapped out by the publisher, who viewed this as a great source of disbelief for the players and consequently having their own disbelief on the publisher, so only eight galaxies were introduced. In order to generate content, and consequent characteristics of the 256 (from the calculation 2^8) planets the game has to offer, a single seed number is created, so that the fixed algorithm knows how many times it should run in order to determine each planet's complete composition (such as position in the galaxy, prices of commodities, and even name and local details). This also takes into consideration that each planet will have different and unique characteristics.

3.1.2 Rogue

Procedural content generation was similarly introduced in games and films, but it was *Rogue* who endorsed the meaning of PCG in games since it was one of its earliest examples, in 1980. After *Rogue*, a series of dungeon crawling games followed the same example, and were then considered *roguelikes*.[6] Amongst the various games created under the influence of *Rogue* and its procedural content generation techniques, we can find *Hack*, *Moria*, *Diablo* and even highly arguably *World of Warcraft*.

[6] Any game based on *Rogue*'s game engine and mechanics, as well as its dungeon game play style.

"Rogue's biggest contribution, and one that still stands out to this day, is that the computer itself generated the adventure in Rogue. Every time you played, you got a new adventure. That's really what made it so popular for all those years in the early eighties." [R. 97]

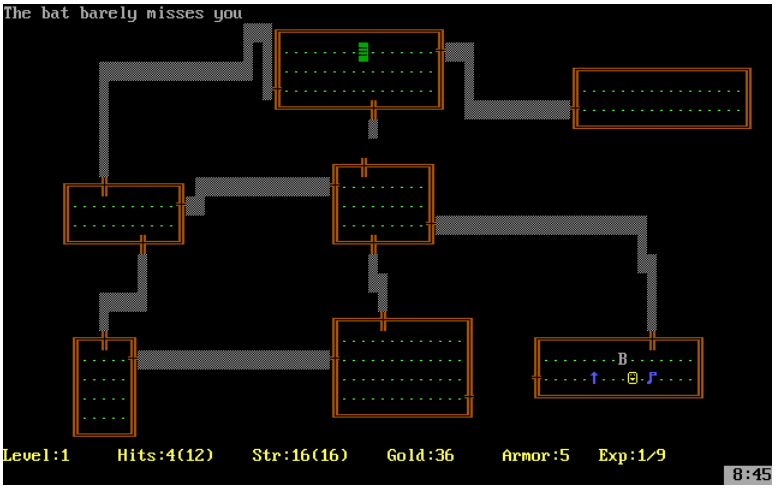

Figure 3.2: Screen shot of the game *Rogue*.

As previously stated, *Rogue* is a dungeon crawling game. The player starts in a dungeon room, and then has to explore it thoroughly in order to find the *Amulet of Yendor*, to then ascend to the surface and exit. Throughout the dungeon, the player will encounter several monsters that will try to stop his demand. Consequently, as these monsters are annihilated, the player gets stronger to withstand the dangers that lie up ahead. The player might not last until the end, though. It is possible that the player will run out of resources (such as food) and die somewhere in the dungeon.

How is *Rogue* using the PCG techniques? The algorithm generates a random new level every time the player descends a set of stairs. [Dou08] Firstly, the map is divided into a grid (3x3 for Rogue generally), and each element will have a flag concerning its connectivity status, as well as an array representing the elements it's connected to. A random element is picked to start with, and is marked as connected. As soon as that is done, the next adjacent element will be picked and marked as connected. This process will be repeated until every single room has at least one connection:

"The elements will then be drawn on the map, as well as their exits, in order to create a corridor that leads onto the next room. Each corridor however, is not linearly-shaped, but "L"-shaped." [Hug]

3.1.3 Slaves to Armok: God of Blood Chapter II: Dwarf Fortress

Slaves to Armok: God of Blood Chapter II: Dwarf Fortress (or simply *Dwarf Fortress*) is a roguelike-game that also focuses on PCG techniques. Starting its development in 2002, and having its alpha release in 2006, the only developer of the project, Tarn Adams, expects the game

to last another twenty years in the main development phase. The game has a great quality for its own kind and can be represented in a grid. However, it revealed itself to be quite complex for any beginner to just play it, compromising its success rate.

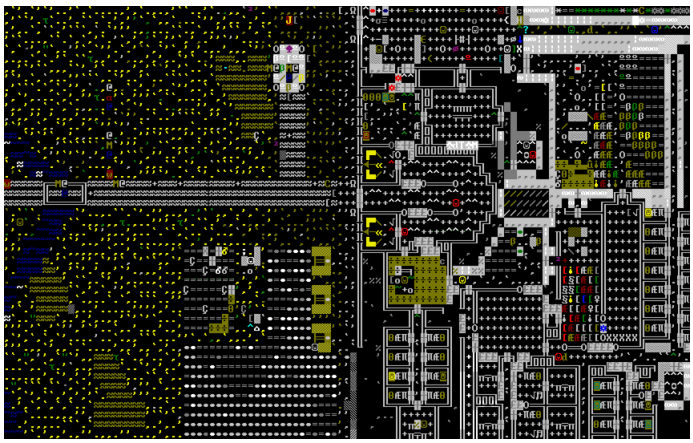

Figure 3.3: Screen shot of the game *Dwarf Fortress*.

The main objective is controlling a civilization of dwarves, and therefore construct a highly-economical, social and strong citadel underneath the mountains. However, the game is highly difficult to master, due to its randomly generated worlds and major mechanics. The game field is constituted of no more than pixel-art elements (as it can be seen in picture 3.3), including Code Page 437[7] characters in various colours.

One of the facts about *Dwarf Fortress* is that not only the levels are generated, but the story as well. As soon as the levels are generated for the player to explore in, the story is also having its facts, events and historical figures documented so that the gameplay experience is enriched. Despite the fact levels are generated randomly according to user input (savagery, size of the world), only one game can occur at once per world. [Dou08] The level generator of Dwarf Fortress works in several layers:

1. *Fractal techniques* (algorithms)[8] - Used to predict the elevation changes of the terrain;

2. *Biome techniques*[9] - Utilized to enhance weather conditions, as well as geographical aspects;

3. *Area definition* - In this third step, areas as defined as good, evil, neutral or benign, wild and savage;

4. *Historical definition* - Placement of population, wild beasts, roads and other life influenced aspects.

[7] Original character set of an IBM computer, or of the MS-DOS system shell.
[8] Algorithms used to create content gradually, in fragments.
[9] Algorithms used in geographical and weather conditions

3.2 Infinite Mario Bros.

Based on the immense blockbuster that *Super Mario Bros.* actually was, and created by the main *Minecraft*'s developer Markus Persson[10], *Infinite Mario Bros.* (IMB) tends to be a complete clone of *Super Mario Bros.* but with a slight difference: all levels are automatically and procedurally generated. This means that the game's controls and mechanics used in IMB are exactly the same as the ones used in *Super Mario Bros.*. However, the graphic style is improved, as well as the levels themselves.

There is no doubt that IMB is a great addition to the pool of games that use PCG as their level construction technique. The mechanism is simple, yet allows the player to have *"infinite" fun*. Every time the player enters a world, there is a generated map with a certain layout, and a certain amount of levels (see figure 3.4). Each level will have its own characteristics built as soon as and every time the player enters it. In addition, every single time the whole set of levels within a world is complete, a totally new world is generated, with a completely different layout, and a theme picked at random.

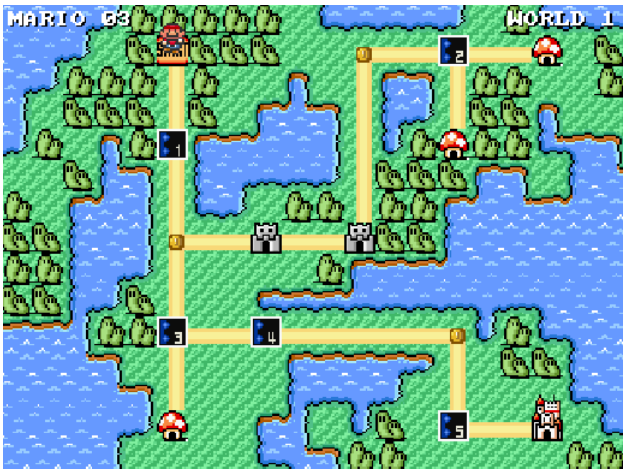

Figure 3.4: Screen shot of a generated world of IMB.

Each level's characteristics are generated upon entry, utilizing a seed that has been previously generated as the game started. These characteristics include the amount of enemies, hills, slopes, pits, blocks to destroy and power-ups, as well as the theme of the levels. Each level has its own theme, which includes background images and background music. As soon as the player is in the level, all these characteristics are generated. Whether the player dies in the level and has to restart it, the engine will re-run the generator and a completely different level will appear, so the player does not feel the monotony and henceforth emphasizing the challenging feel of the game.

So, in short, the game is completely never ending:

> *"No textual description can fully convey the gameplay of a particular game"*. [STY+11]

[10] Known as *Notch*.

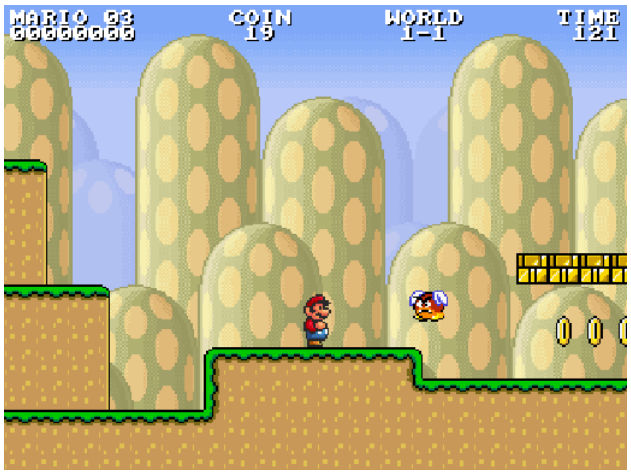

Figure 3.5: Screen shot of a generated level of IMB.

Infinite Mario Bros. is certainly a great addition, as it creates entertaining experiences throughout its gameplay time.

3.3 The Mario AI Championship

The *Mario AI Championship* is a series of events co-organized by *IEEE*, that is aimed at developers' skills to create levels using procedural content generation, through an adaptation of the *Infinite Mario Bros.*' code. The competition has been around ever since 2009, every single summer. However, in the years 2011 and 2012, the number of entries failed to complete the actual minimum (five entries). [TSKY12]

3.3.1 Divisory

Ever since 2010, the *Mario AI Championship* grew to the point of pertaining up to *four* different tracks (in this case, sections). Each one of these tracks considers a different area of expertise, and therefore different (or in some cases the same) participants. These four areas were described as follows [KT12]:

- *Gameplay track* - Deriving from the the 2009 Championship, the gameplay track had its objective relying on the ability of contestants to create agents that were capable of completing as many levels as possible, considering speed and skill characteristics. In this section, it is normal to see entries that utilize algorithms such as A*[11], or even rule-based algorithms, utilizing architectures more commonly known as *reactive architectures*[12];

- *Learning track* - This section pertained to the entries which contained learning agents. These agents had the sole objective of acquiring knowledge as they progressed through

[11] A-star algorithm can be defined as a path-finding algorithm.
[12] Architectures designed to hold agents that would need a fast reaction timespan.

the level several time, so agents with a greater learning advantage would sum a higher value of points. In most cases, neural network and genetic strategies would be adopted throughout the agents' creation, invigorating the contestants' capabilities of creating agents that learn with time;

- *Turing test track* - Many of the entries on the previous two tracks did not consider their human-like nature, since most of their behaviours were humanely impossible to achieve. Keeping the idea of human-like agents, the *turing test track* has the sole objective of creating artificial intelligence agents that behave like humans, so in the end a normal person would never distinguish their true artificial nature. With this in mind, the Turing test track aims to record gameplay videos of agents and consider their human characteristics while progressing through a level [STY+13];

- *Level generation track* - The one and only track obtaining greatest focus in this dissertation, and therefore being the most considered for the project itself, the level generation one has the sole objective of evaluating how well a level generator creates new, innovative and fun levels. Even though fun is taken into account, participants tend to ignore this requirement and focus on creating level generators that consider performance as their greatest objective.

Considering the various tracks on the *Mario AI Championship*, the next chapter will describe some of the entries on the Mario AI championship throughout the years 2009 and 2010.

3.3.2 Sample entries

3.3.2.1 Gameplay track

In 2009, Robin Baumgarten participated in the game generation track of the *Mario AI Championship*, thus presenting a fast (if not the fastest) way to completing a set of levels within the *Infinite Mario Bros.* game. Applying the A* algorithm, Baumgarten was able to make an agent go through a level as fast as possible, utilizing only the techniques of movement, speed and jumping.

However, the technique was not simply based on applying the A* algorithm, but as well as the surrounding environment. Basically the agent would have to analyse all of the possible paths (including jump sections or obstacles) ever since its standing point (current node) up to the rightmost edge of the screen (end node), and decide which one would allow it to complete said obstacle in the shortest amount of time. Since the objective of the algorithm is exactly that, finding the shortest path possible, there are no extra regards to the techniques applied other than when to accelerate, jump or shoot fireballs (if *Mario* is in the correct state to do so). [BC10]

It is great to note that amongst the 15 competitors, the ones that scored the most were the A* users. Amongst other entries, the following ones were rule-based algorithms to complete the levels, mostly considering rules of when to jump, accelerate or shoot fireballs. Other formalities were presented throughout genetic programming or state machines, which allowed the plumber to complete the levels.

3.3.2.2 Level generation track

Winner of the 2010 competition's level generation track, Ben Weber developed a generator suiting the characteristics and ideals of probabilities and step-by-step programming. His generator based itself in creating highly customizable levels, therefore adjusting them to the players' needs. *Probabilistic multi-pass generator* surpasses simplicity, and provides interactive levels.

Consisting of six phases, the generation enforces two types of constraints pertaining playability and competition issues. Each of these six phases will create different fragments and features of said level, while utilizing probability calculations [STY+11] (see figure 3.6):

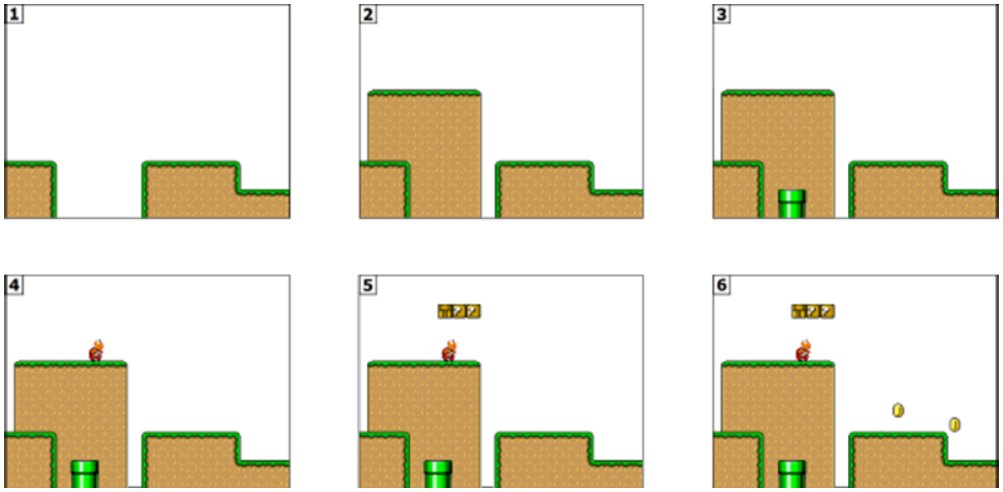

Figure 3.6: The six phases allocated by ProMP Generator.

- *Ground* - First and foremost, the generator considers the ground in which the character will walk in. This phase also considers the amount of gaps to be calculated;
- *Hills* - In another phase, the generator considers the extra hills (higher grounds) that the level would contain, so that the platforming originality is maintained;
- *Pipes* - Pipes are utilized by *Mario* to navigate through the level into secret areas, or even through the level if in any case. However, they can also be some obstacles to completing the level, so they are added;
- *Enemies* - One of the most important parts to include inside a platformer's level. Enemies are placed accordingly to the parameters introduced by the user, such as enemy density (quantity of enemies in a certain amount of space), or even enemy quantity;
- *Blocks* - These items represent squares on the level that contain Mario's *power-ups*, such as mushrooms or fire flowers;
- *Coins* - An extra addition to the field, allowing the player to augment their score.

Each and every single one of these items must therefore be included in the level's field, and mandatorily regulated by the user upon generation. Each user and/or player must define how the level will be in terms of content and features, since this generator does not support any kind of *offline training*[13].

As Ben Weber was not the only participant of the 2010 competition's level generation track, a group of contestants came together to create *The Hopper Level Generator*. This generator, developed by Glen Takahashi and Gillian Smith, proposes a player adaptation according to their playing style, skills and occurrences during game play, whilst taking advantage of the usage of probability calculations.

Most of the level (at least 85%) is generated at first. All of the generated content is generated throughout probability calculations according to the player's progress. Since the first generation will not take those aspects into account, most of the firstly generated content is random, but logically oriented to an *easy* and low difficulty approach, pertaining to a short amount of gaps. However, difficulty is not the only aspect considered, since the generator also adapts itself to the player. Similarly, the number of hills and obstacles will augment with the player's tendency to jump. With this, three main player skills are considered [STY+11]:

- *Speed runner* - Whether the player speeds throughout the level. He will be automatically considered a speed runner if such happens, and therefore have the level suited according to its characteristics (less gaps, less hills and pipes);

- *Enemy killer* - Another style corresponds to the player that takes his time to eliminate enemies on the level, so great amount of the level's features will include an augmented number of opposers;

- *Explorer* - Considering the exploring nature of the player, this style corresponds to the one that usually goes through the level to discover its secrets and the hidden *power-ups*, so the numbers of these items will be augmented accordingly.

In addition, all of the above skills are not mutually exclusive. A player can have all three skills or game play styles simultaneously, therefore making the generator's output a lot more interesting and competitive, as well as the difficulty of said levels. Whether the player died whilst completing the first level or not, the generator will take this in consideration to make understand the player's difficulty level, *easy*, *medium* and *hard*.

However, the generator isn't as simple as creating the level and adapting to the player's conditions. Alas, the remaining 15% can contain secret pre-made areas by the programmers. Some level fragments have been designed to fit into any part of the probability-based generated levels. These areas can be described as in *hidden coin zone*, *fire zone*, *shell zone* and *super jump zone* (see figure 3.7).

[13] AI Agents training throughout player's experiences, in this case the level generation agent will adapt to the player's skills.

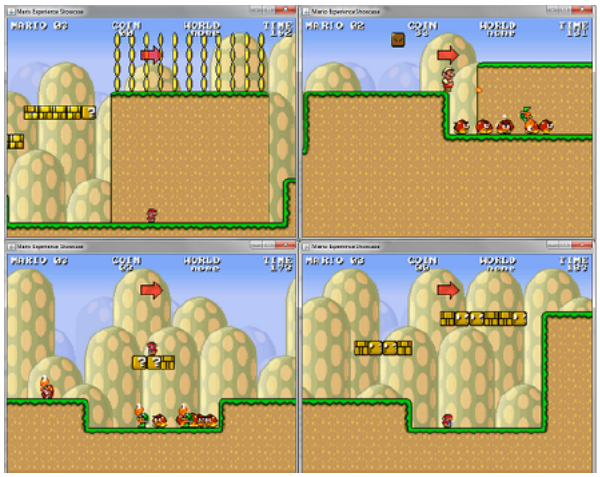

Figure 3.7: Hidden generated zones on *The Hopper Level Generator*: *hidden coin zone*, *fire zone*, *shell zone* and *super jump zone*, from top to left.

3.4 Procedural Content Generation - Genetic Algorithms

Still in the procedural content general domain, apart from the algorithms described above, we can find genetic algorithms to generate levels automatically, turning them into some kind of evolutionary methods to create new and innovative levels. As the name also implies, these algorithms are based on Darwin's theory of Natural Selection, where each individual being will fit their characteristics according to the environment around them. [MdSB11] Another of their characteristics allows an *Individual* to be defined, coded with specific data and features. These features will allow this *Individual* to be scored, and therefore compared with other *Individuals* in order to understand if they pass on to the next generation of the algorithm, normally through a *Fitness Function*.

Applying this approach to level generation can be simple, since each *Individual* will represent a level on its own. Each level is divided with a grid, containing cells. Each one of these cells will contain information on a tile of the level, or any kind of particular information that might be relevant to be kept in the cell itself. It is important to understand how these cells will generate content. First and foremost, the *fitness function* will consider a plausible and possible path structure. After that has been set, the cells can be filled in via their individual analysis, which will decide what content to be placed in the grids, as well as their surrounding ones. As soon as the path is completely applied, it is time to ensure its smoothness, as well as its aesthetic appearance. It is imperative that the level actually makes sense on its own, that it considers a possible path, and that all grids are successfully filled. After the fitness function has finished its work, two extra techniques will be used to generate content: *Mutation* and *Crossover*. [MdSB11]

- *Mutation* - Consists of applying changes to random cells, so that the levels will have a higher diversity rate. This is applied throughout probabilities, and random algorithms;
- *Crossover* - Consisting on the crossing of two *Individuals* to create a new one, with the characteristics of both. These new *Individuals* will have the qualities of the ones used to

merge, as well as all the coherence and playability.

In order to finalize all of these techniques and their fused work, it is important to revise the adequacy and coherence of the levels created. With this comes the necessity of adding extra features to the levels, such as traps and/or enemies, so immersion can achieve levels that are expected by players.

3.5 Chapter Summary

The present chapter presented most of the related work within the *procedural content generation* area, such as the games *Elite*, *Rogue* and *Dwarf Fortress*. These games were described alongside *Infinite Mario Bros.*, a remake of the classic *Super Mario Bros.*, but with infinite levels. Still connected to the *Infinite Mario Bros.* game, the *Mario AI Championship* is also described, within its competitions, tournaments and some of the best entries throughout the years. Finally, an overview on *genetic algorithms* utilized in some game level generators, as wlel as their corresponding *fitness functions*.

Chapter 4

Level Generator Platform

The following chapter presents the organization and architecture of the level generator platform. The created application allows for the users to prompt several parameters that influence the type of levels that are generated. Each level is then customizable according to any user's likings, as well as according to the user's skills. With that in mind, several *modules* have been implemented, such as *Parametrisation*, *Level Creation*, *Level Gameplay* and *Level Management*. The life cycle of the program can be found in Figure 4.1.

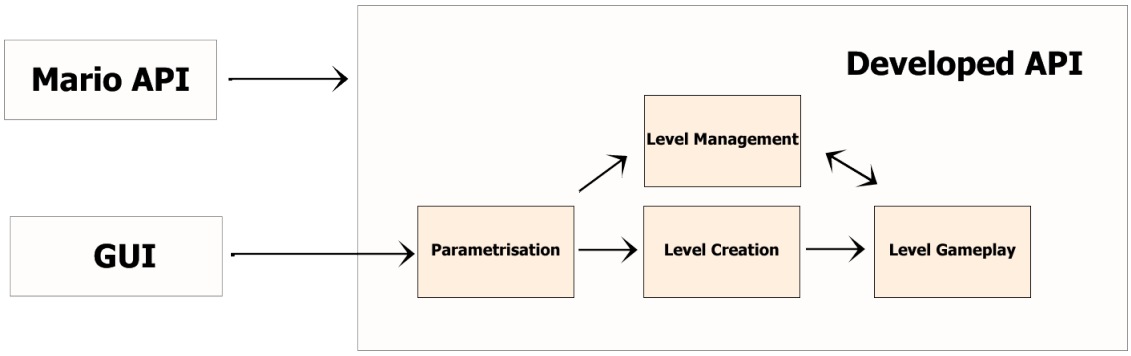

Figure 4.1: Level generator platform's life cycle.

4.1 Parametrisation

The *Parametrisation* module emphasis on gathering data from the user to generate entertaining levels. Each one of the parameters needed for the generator to complete its task are elicited in this module.

As it can be seen in Figure 4.2, the parameter selection window is sectioned into two parts: *Level Properties* and *Probability Properties*.

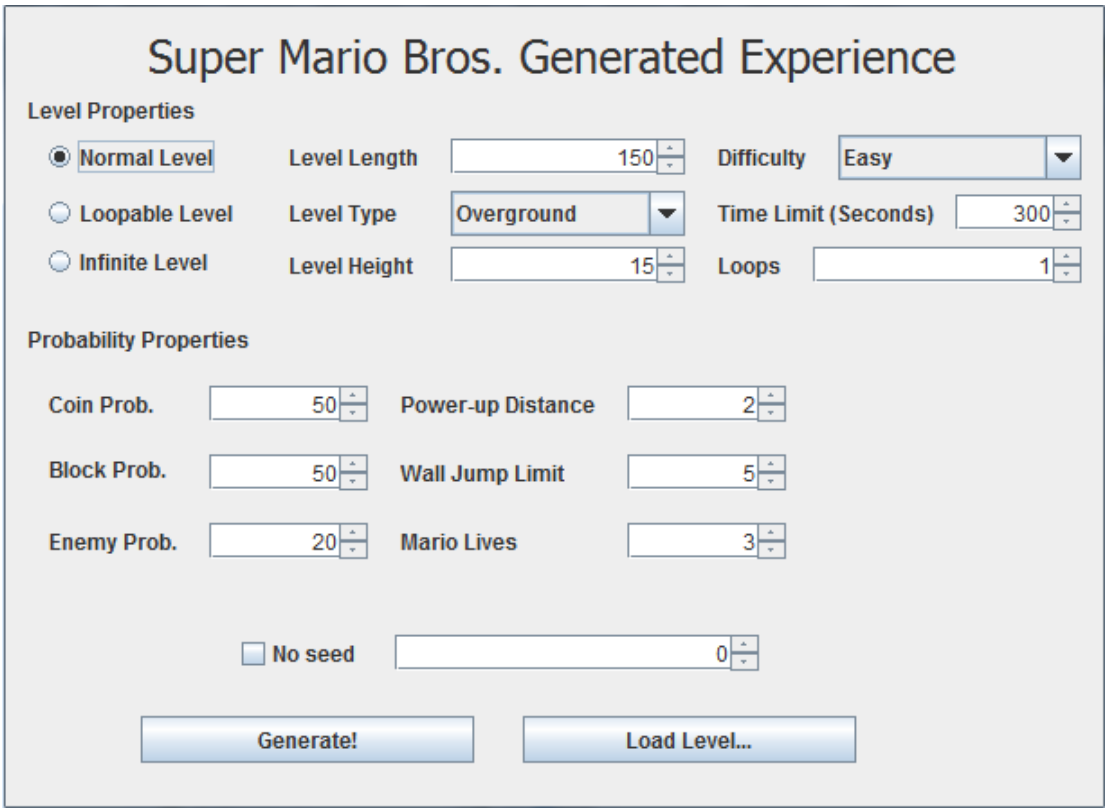

Figure 4.2: Parametrisation of the level generator.

4.1.1 Level Features

In order to create entertaining levels, each level requires several aspects. In this section of the parameters selection, the visual aspects of the level and its design will be taken into account.

- **Level kind** - Users can choose from three kinds of levels: *normal*, *loopable* and *infinite*. This parameter influences the length of the level, and how long it might become.
  - **Normal level** - A normal level follows all of the parameters directly, and creates a level with the assigned length.
  - **Loopable level** - With the objective of creating interactive levels, this kind of level follows the number of loops assigned to the generator. Every time the user reaches half of the level, the generator will create a different level portion and add it to the end of the current level. The process is repeated as many times as the number of loops assigned.
  - **Infinite level** - As the name indicates, this kind of level is never ending. The assigned level length is taken into consideration, as it first creates a level with that very same length. Similarly to the *loopable* approach, as soon as the user reaches half of the current level, a different level portion containing the same length will be attached to the end, disregarding the number of loops assigned and creating a never-ending level.

- **Level Length** - Every single level generated will have an assigned length. The spinner[14] allows the user to perform said selection. Each *spin* of the box increments or decrements by 15 (fifteen). The selected length will be differently applied according to the level *level kind* selected. Normally, the level length is as defined. However, for the *loopable* and *infinite* approaches, the length determines how long each generated portion will be.
- **Level type** - The level type takes into consideration the aspect of the level. Each type has a different layout (refer to Figure 4.4, providing a different insight into the level as far as users are concerned. The choice is bound to three types:
  - **Overground** - Overground is a level conducted outdoors, on the main environment of the *Super Mario Bros.* world. The background includes a blue sky, and some other landscape features such as trees, or hills as it can be seen in Figure 4.4. In addition, this is the only type of level the user will find hills on.
  - **Underground** - As not all levels are overground, this kind of level is held under the ground, enabling the player a deeper and darker experience. The background becomes brown, and has cave-style patterns (refer to Figure 4.4).
  - **Castle** - Based on the classic *Super Mario Bros.* layouts, castles represent the last levels of each of the worlds in the game's universe, where the player would be confronted with the villain *Bowser*. This type of level is a recreation of those layouts, in which the background becomes black, and the walls gather up like stones.
- **Level Height** - Every single level generated will have an assigned height as well. This spinner allows the user to selected the preferred height, being the minimum required height of the level 15 (fifteen).
- **Level difficulty** - Each one of the generated levels contains a selected difficulty, adjusted to each user's preferences. There are three difficulties that can be chosen from: *Easy*, *Normal* and *Hard*.
- **Time limit** - A *platformer* includes the timer mechanic, so consequently, these *platformer* generated levels also do so. The user can define a time limit (or no limit at all), in seconds, to complete the levels.
- **Loops** - Utilized by the *loopable* level function, this spinner allows the user to control how many portions of the level will be dynamically created by the generator while playing a *loopable* level, being this number defined by the *Loops* parameter.

4.1.2 Probability Properties

As the level is not only made out of the blocks, there are also the elements. This section of the parameter selection allow the user to choose the probability of certain aspects of the level being created.

[14] A spinner is a box that allows the user to select a number, or to *spin* around the allowed numbers within that box.

- **Coin probability** - One of the elements of the game is the *coin*. Each coin is collectible as Mario touches it. This probability will allow coins to be created with the selected percentage in each portion of level (refer to Section 5.2.2 for more information on how the levels are divided).
- **Block probability** - Blocks also play an important part in the game, since they allow Mario to gather more collectibles, or even to get stronger. Following the same lines as the coin probability percentage, the block probability is also a percentage which will allow creation of blocks within each of the level's portions.
- **Enemy probability** - As far as difficulty is concerned, enemies are also taken into account to make the user's life harder. This probability percentage will determine how many enemies will be spawned in each portion of the level.
- **Power-up Distance** - There is a slight chance that each one of the blocks contains a power-up. However, they might be too close to each other. This parameter allows the user to set how distanced the power-up will be in throughout the level's portions.
- **Wall Jump Limit** - Within each level, there is the chance of one of the portions containing the *Wall Jump* area (refer to Section 4.3.6). The user can then set a limit of how many *Wall Jumps* will be rendered into the level.
- **Mario Lives** - Users have several tries to complete levels. With this parameter, the user can set how many tries he/she is going to have in order to complete the generated levels.
- **Seed** - Used to feed the random generator, mainly used for debugging purposes. The user can choose not to have a seed[15] (in this case, a temporal seed shall be created);.

4.2 Level Creation

The objective of the level generator is to create levels, and create new and entertaining dynamic content. Each one of the levels is divided into several portions, named *Cells* (refer to Section 5.2.2). For each one of the *Cells*, content will be created, utilizing *level elements* to compose *level zones*. Every game is characterized by its visual style, enhancing the user's affinity to the game. [FF01] As this game follows the *Infinite Mario Bros.* example regarding its graphical content, the set of figures presented afterwards represent the graphical icons of the *Mario* universe.

4.2.1 Mario

The character is of utmost importance in order to play the generated levels, else they would never be playable at all. *Mario* has different states (refer to Section 2.2): *Small*, *Big* and *Fire*. Figure 4.3 illustrates *Mario*'s different sprites[16] for each one these states.

[15] Number utilized to initialize a random number generator.
[16] A sprite is a representation of various movements performed by 2D graphical art in an animation.

Figure 4.3: *Mario's* states, *Small*, *Big* and *Fire*, respectively.

Each one of the sprites represents one portion of *Mario*'s movements. For each action performed, a set of sprites will be triggered in order to present that very same action in its totality.

4.2.2 Level blocks

A level block represents a square on the screen, sized 16 pixels per 16 pixels (as shown in Figures 4.4 and 4.5). Each one of these blocks is responsible for the level's design, and how it looks. However, most of the blocks change according to the type of level generated. Each type of block has its own value in the project developed, assigned as *byte* values. Each one of the values deports a calculation to access the position of the square in the block information file. This calculation is based on the number 16, since each square is 16 pixels wide and 16 pixels tall.

Figure 4.4: Blocks that represent *Ground*, *Overground*, *Castle* and *Underground*, respectively.

Each type of level has its own block design, being different in their core styles. However, the type *Overground* is the only type of level that can contain *grass*, *hills* and *tubes*. This means that for the types *Castle* and *Underground* all of the level blocks that contain information on *grass*, *hills* and tubes will not be used in these levels.

There are however, blocks that are used in any generated level. Blocks such as *rocks*, *coins*, *brick blocks*, *coin blocks* and *power-up blocks*. These blocks are merely decorative or collectible by the player. An extra set of blocks are the *exit blocks*. In every generated level, the *exit blocks* mark the end of the level.

Figure 4.5: Blocks that are usable in any type of level, with the exception of the *tubes*.

4.2.3 Enemies

Another level element that is used to complete the scenery is the *enemy* element. Each enemy has its own characteristics and way of acting, as well as its own *byte* values. *Mario*'s task is to overcome the enemies and proceed to the end of the level. Most of the enemies can just simply be jumped on, others will need to be hit with objects or *fireballs* (refer to *Mario*'s *Fire* state, in Section 2.2), or even need two hits. Enemies can then be divided into sections:

Figure 4.6: Enemy sprites in the world of *Mario*.

- *Goomba* - *Goombas* are simple enemies that look like balls, and have feet. These enemies run around the platforms they're inserted in, and don't stop by the ledges. In case *Mario* runs against them, the player will then lose a try.
- *Koopas* - These enemies are basically turtles which when hit hide inside their shell. In order to dispose of these enemies, *Mario* must hit them from the top, and then kick their shells into a hole, or have two *Koopas*' shells colliding with each other. *Mario* also has the power to grab a shell as soon as the enemy is hidden inside. There are however, two kinds of *Koopas*: Green and Red. The only difference is that green ones fall from ledges, while red ones do not.
- *Bullet Bill* - These flying bullets cross the screen from side to side, on an horizontal manner. They just follow a planned route, so there are no effects considering ledges.
- *Chomping Flower* - Flowers are enemies that are part of the level's soil or tubes. They have the ability to jump when *Mario* is close by, forcing him to either way jump over them or pass under them. They cannot be knocked off by jumping on top of them.
- *Spiked Koopas* - *Spiked Koopas* are another kind of *Koopa*. Despite being small, these enemies contain spikes on their shells, protecting them from direct hits from *Mario*, be it by jumping on them or being hit with *fireballs*.
- *Winged Enemies* - Any of the enemies referred above can be *winged*, except for *Bullet Bills* and *Chomping Flowers*. Wings allow the enemies to jump at will, making *Mario*'s task of completing each level harder.

4.3 Zones

Each one of the generated level's portions will contain a specific level zone. Each one of these level zones have their own characteristics, and differ from one another in *style*[17]. Some zones require more skill, while others can be simply crossed with a faster pace. The zones available for this project are *Straight*, *Jump*, *Hill*, *Dunking*, *Manoeuvre* and *Wall Jump*.

4.3.1 Straight

With the need of a simpler zone, the *Straight* zone comes in hand. As shown in Figure 4.7 It simply contains *ground* blocks up to a certain height on game screen, and follows that height until the end of the level's portion. This zone is used in specific situations such as the start and the end of the level, in order to simplify the player's entry on the game, as well as simplify the exit blocks' building process.

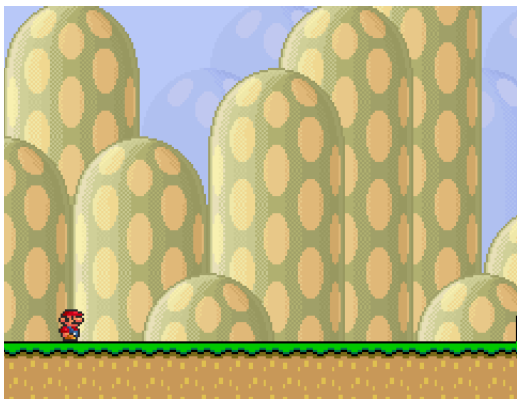

Figure 4.7: Example of a straight zone.

4.3.2 Jump

Similarly to the *Straight* zone, the *Jump* zone is composed of blocks up to certain height of the game screen, following a straight line until the end of the level's portion. However, *Jump* zones contain a gap in between, which is an absence of blocks in order to create a zone that *Mario* must jump over. As an extra element, *rock* blocks can be places at the edges of both sides of the gap, with a certain probability. The final result is then presented in Figure 4.8. This allows the player to put an extra effort to jump over the gap, regardless of it being large or not.

4.3.3 Hill

Hills are more of a decorative zone, but can also be considered item collecting zones due to their landscape changes. These zones are composed of a basic *Straight* zone with the capability of constructing higher landscape portions (refer to Figure 4.9). These portions are constituted by *hill*

[17] Each player has his/her own gameplay style, and that influences on which zones they will be better or worse at.

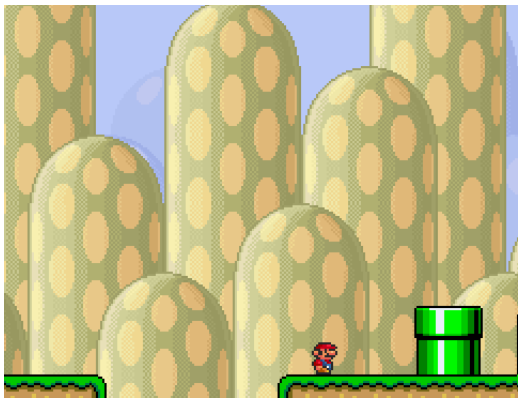

Figure 4.8: Example of a jump zone.

top blocks on the highest portion of the landscape, *hill side* blocks on the ledges and *hill fill* blocks filling the landscape's interior. All of these generated *hills* automatically become a higher piece of floor, where *Mario* can walk or dodge enemies.

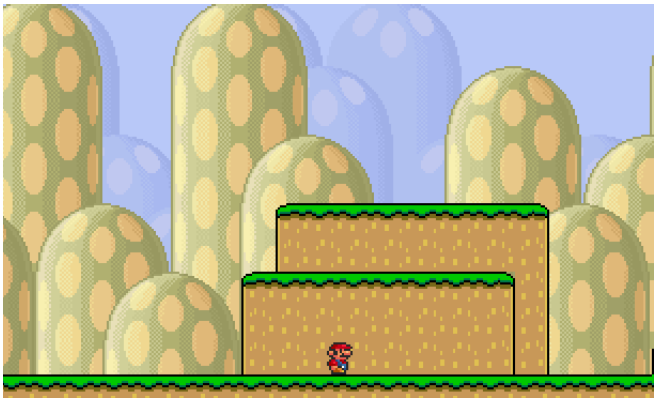

Figure 4.9: Example of a hill zone.

4.3.4 Dunking

From the need to implement an area where players would obtain a larger quantity of enemies to kill (refer to Section 6.1, where the responses to the first social experiment reveal that people like enemies the most), the *dunking* zone was created (refer to Figure 4.10). These zones are simply *straight* zones slightly lower than the previous generated zone, but with an augmented quantity of enemies. These enemies are selected at random, and have no connection in between any of them. However, they are only accessible when levels with *normal* difficulty are chosen.

4.3.5 Manoeuvre

Accessible on from playing levels under the *hard* difficulty, these zones were implemented with the objective of generating zones that would put the players' skills to test. As any other zone, the *Straight* zone functions as a basic holder, where a gap is added. *Rock* blocks are then placed it on

Figure 4.10: Example of a dunking zone.

the utmost end of the zone, forming a ladder in order to have players use their skills to overcome the obstacles, as shown in Figure 4.11. Mostly, when these zones have a large number of enemies, they automatically become harder to overcome, since the player will have to avoid the enemies.

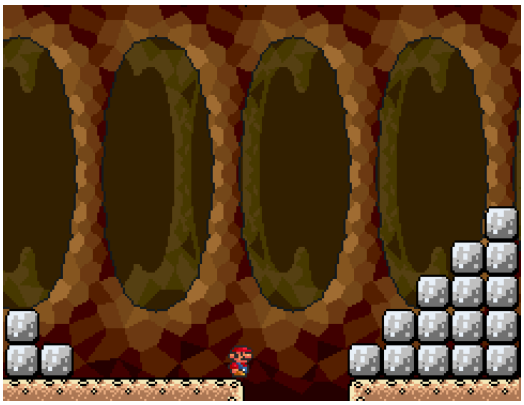

Figure 4.11: Example of a manoeuvre zone.

4.3.6 Wall Jump

The most innovative zone of the level generator is the *Wall Jump* zone, which can only be found in *hard* levels. Unlike the traditional *Super Mario Bros.*, this version of the game allows the player to jump from a wall to another, and out of the wall's ledges. These zones will reflect said mechanic, and will enhance interactivity from players towards the environment. The landscape is organized with two extra platforms, composed solely of *ground* blocks. At the utmost end of the level's portion, a wall of blocks created. Set aside from it, there is an helping platform hanging on the air, which will allow the player to interact the previously created wall, thus activating the ability to jump from wall to wall (refer to Figure 4.12).

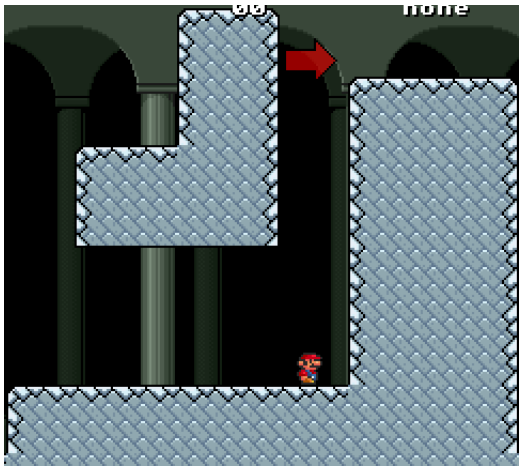

Figure 4.12: Example of a wall jump zone.

4.4 Level Gameplay

The generated levels are inserted into the game scene, and then are eligible to be played. However, a player can control *Mario* easily, and see his movements as he progresses through the level. To control *Mario*, the following keys are of utmost need:

- *Arrow keys* - These keys are needed to move *Mario*, towards the left or right directions;
- *A* - The *A* key allows *Mario* to become faster. This will allow the player to surpass certain obstacles;
- *S* - The *S* key makes *Mario* jump from the ground. This key can be used to jump over gaps or other obstacles.

According to Figure 4.13, the level scenario displayed shows all of the information needed for the game's flow.

1. The character the player will play with, *Mario*.
2. Amount of lives (tries) the player has left.
3. Amount of coins attained throughout the level.
4. World the player is currently in. For the generator, the world will always be blank, since all of the generated levels do not represent a storyline.
5. The amount of time the player has left to conclude the level.
6. Level elements such as blocks and coins, present in the world of *Mario*.

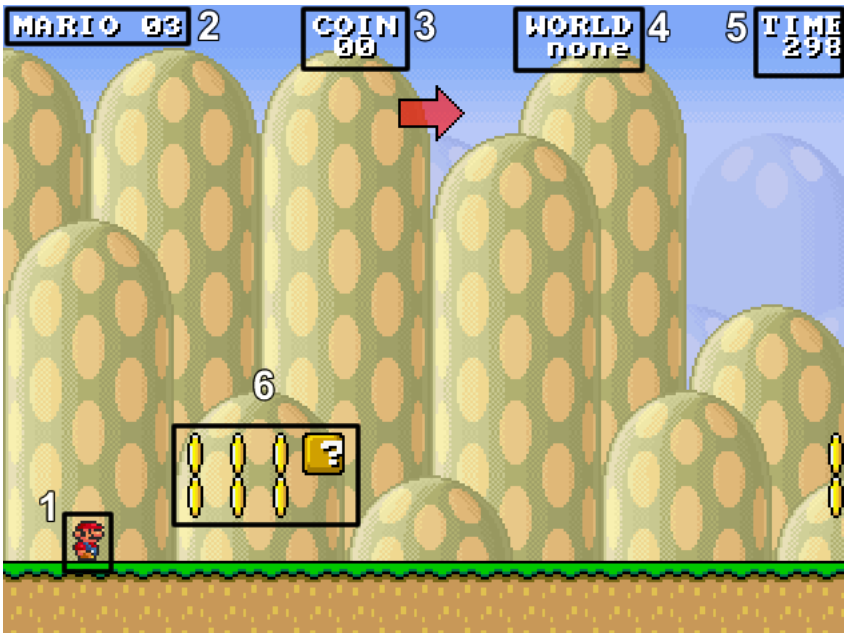

Figure 4.13: The level scenario displayed by the level generator.

4.5 Level Management

The module of *Level Management* allows the user to control which levels he/she wants to play, by using the *Load* and *Save* features. These features will open new doors for players to understand which levels they want to keep and play on another time.

4.5.1 Save

The *Save* feature, as the name states, is a feature that enables users to save the levels that have been generated. Every time a level is generated, a frame will be visible with a text field and a button that will allow the user to write the name of a file, and then save the level. This level will be stored with the written named, attached to the *.mario* extension. The generator saves, in this file, data that is pertinent to the construction of the level, such as the *Cells* and their respective content, as well as information such as the level's difficulty or type.

4.5.2 Load

Opposing the *Save* feature, *Load* enables users to obtain the previously saved levels, and then allow them to play those levels. It retrieves all of the information regarding the level's content, and its vital characteristics such as difficulty and type. A level can be loaded as many times as the user wishes to.

4.6 Chapter Summary

In this chapter, an overview on the level generator has been presented, such as its functionalities and features. Firstly, the parametrisation of level features and its probabilities is of importance, since these are the values that will drive the level generation. Even though these are elements that are introduced by the users according to their preferences, the level elements and consequently zones are created by the generator. Each one of the zones contains its own set of elements, presenting itself differently and uniquely from all of the others towards the player. However, in order to play the game, several keys must be used, and the game scenario must be comprehended, added to the fact that each player can *save* or *load* levels at will.

Chapter 5

Implementation

As it was explained previously on Chapter 4, the *Level Generator Platform* has certain rules that have been applied throughout the implementation process. This chapter aims to detail the implementation process throughout the development life cycle, as well as all of the techniques utilized.

5.1 General Overview

Most of the implementation process derived from the *level generator source code* of the *Mario AI Championship*. The *source* was organized into several modules that constitute the game play, the assets and the mechanics of the game. However, the main task of the project was to implement a level generator within the *CustomizedLevel* class. Figure 5.1 shows an overview of the system's *source code*'s.

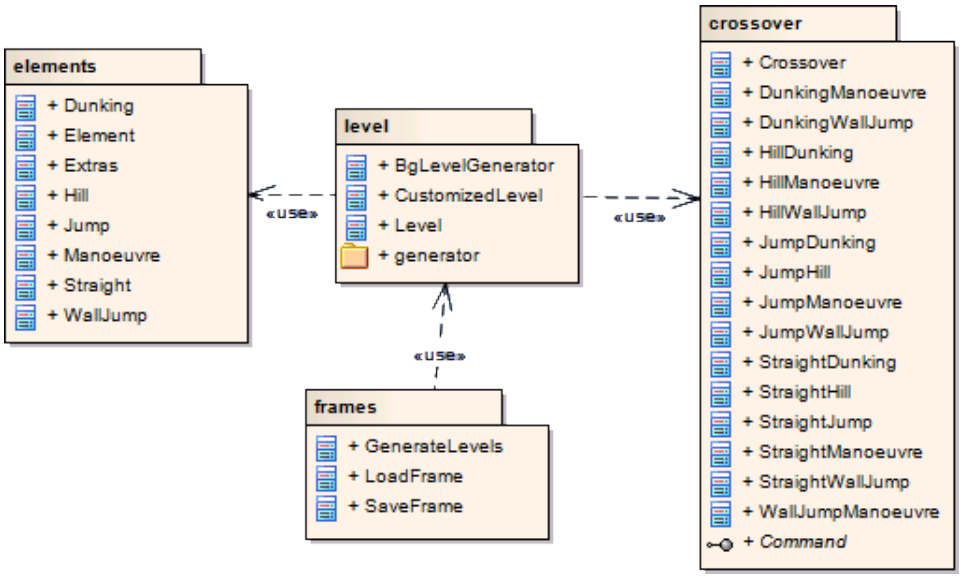

Figure 5.1: Modules developed for the level generator.

- **frames** - The package *frames* is the one that initializes the needed frames[18] for the generator, such as the *parametrisation*, *save* and *load* frames;
- **level** - This package will then include all of the needed classes in order to create levels, and all of the methods utilized by the generator to take decisions relating said generation;
- **elements** - Package that includes all of the level elements, as well as other methods that will be utilized to enhance the level's quality, such as *Extras*;
- **crossover** - The package at hand will then contain all of the information relating the *crossover* technique (refer to Section 5.2.5). This information will then be utilized by the generator in a later state.

5.2 The level generation

The generation of the level becomes crucial in the life cycle of the program's execution. In this chapter, the techniques utilized will be described in order to deeply understand each one of the mechanics that have been implemented during the process.

5.2.1 Constraints

Each one of the generated levels has several constraints which are defined before the generation takes place. These constraints are based on characteristics of the level, such as height, length, coin and block placement or any other value that might become relevant to constrict the generation according to the user's likings. Listing 5.1 shows most of the variables that have been used to constrict the game.

```
1  MINIMUM_DISTANCE = 15;
2
3  MAX_GAP_LENGTH = 4; MAX_GAP_SAFEZONE_LENGTH = 4; MAX_STAIR = 2;
4
5  MAX_FLOOR_HEIGHT = 4; MIN_FLOOR_HEIGHT = 1;
6  MAX_GAP_BETWEEN_HILLS = 3; MIN_GAP_BETWEEN_HILLS = 2;
7  MAX_HILL_HEIGHT = 11; MIN_HILL_HEIGHT = 15 - MAX_HILL_HEIGHT;
8
9  MAX_DIST_COIN_FLOOR = 3; MIN_DIST_COIN_FLOOR = 2;
10 MAX_LINES_COINS_VERTICALLY = 3;
11
12 MAX_DIST_BLOCK_FLOOR = 4; MIN_DIST_BLOCK_FLOOR = 3;
13 MIN_POWERUPS = 3;
14
15 MAX_TUBE_PROB = 2; MAX_TUBES_PER_CELL = 3;
16 MIN_GAP_BETWEEN_TUBES = 2; MAX_TUBE_HEIGHT = 2;
17
```

[18] Window that is presented to the user.

```
18 WINGED_PROB = 5; DUNKING_ENEMY_PROB = 2;
```

Listing 5.1: Cell constraints applied.

- **MINIMUM_DISTANCE** - This value decides the length of one of the level's portions (refer to Section 5.2.2);
- **MAX_GAP_LENGTH** - The maximum length of a gap in a *Jump* zone (refer to Section 4.3.2;
- **MAX_GAP_SAFEZONE_LENGTH** - The maximum length of the safe zone within each *Jump* section (refer to Section 4.3.2;
- **MAX_STAIR** - The maximum amount of *rock* blocks for the stwairwell of a *Jump* zone (refer to Section 4.3.2;
- **MAX_FLOOR_HEIGHT** and **MIN_FLOOR_HEIGHT** - The maximum and minimum amounts for the level's floor height. The height is counted from bottom to top;
- **MAX_GAP_BETWEEN_HILLS** and **MIN_GAP_BETWEEN_HILLS** - The maximum and minimum distance values between hills;
- **MAX_HILL_HEIGHT** and **MIN_HILL_HEIGHT** - The maximum and minimum height values for hills (refer to Section 4.3.3;
- **MAX_DIST_COIN_FLOOR** and **MIN_DIST_COIN_FLOOR** - The maximum and minimum distance values for coins from the ground;
- **MAX_LINES_COIN_VERTICALLY** - The maximum amount of lines allowed for the placement of coins. Each coin block will have this maximum amount of lines vertically;
- **MAX_DIST_BLOCK_FLOOR** and **MIN_DIST_BLOCK_FLOOR** - Similar to the coins, *item blocks* will also have a maximum and minimum distance;
- **MIN_POWERUPS** - Minimum amount of *power-ups* throughout the whole level;
- **MAX_TUBE_PROB** - The maximum probability value for tube creation;
- **MAX_TUBES_PER_CELL** - The maximum amount of tubes a level portion (refer to Section 5.2.2) can contain;
- **MIN_GAP_BETWEEN_TUBES** - The minimum distance between two tubes within a level;
- **MAX_TUBE_HEIGHT** - The maximum height allowed for tubes;
- **WINGED_PROB** - Probability for an enemy to be spawned with wings;

- **DUNKING_ENEMY_PROB** - Probability of creating enemies in a *Dunking* zone (refer to Section 4.3.4);

5.2.2 Level Portions

Each level is divided into several portions[19], allowing the content to be generated into any of these portions individually. A portion has been given the name *Cell*, and has the proportions of 15 squares of height by 15 squares of width. This amount of squares was selected due to the fact that a portion should not be too big or small, also making the level division an easier task. Within each one of these *Cells*, content can be placed onto the *Element* object, which contains all of the information pertaining that portion of the level. Each *Element* will then contain coordinate and block informations, as well as which content has been assigned to this portion. The level portion contains the following information:

- **xStart** and *xEnd* - These variables contain the coordinates in which this level's portion starts and ends;
- **maxFloor** - A variable which stores the maximum height of the floor for this level's portion[20];
- **levelHeight** - The current level's height is stored in this variable, for further needed calculations;
- **floorHeight** and *realFloor* - Two variables that serve almost the same purpose, but end up being used in different contexts. *floorHeight* is utilized to understand where assets should be placed throughout the level's portion, whereas *realFloor* is utilized to place any kind of objects onto the floor, and which is the real value of the floor's height;
- **blocks** - This *list* contains all of the information relating to the blocks contained in this level's portion, utilizing the *Block* object;
- **levelPiece** - Each *Cell* will have content, and this object allows the generator to save which zone this portion is referring to. Utilizing the *CellContent enum* (refer to Section 5.2.4), this information is kept accordingly;
- **groundFloor** - The *HashMap*[21] stores information relating to which is the height of the floor in each one of the X coordinates of the element. This object serves the purpose of understanding which is the height in each single coordinate in order to place relevant objects;
- **combinationMethod** - In order to understand which *crossover* technique has been utilized for this portion of the level (refer to Section 5.2.5), a *String* becomes useful to store the technique utilized.

[19] Sections of the level.
[20] The value assigned is calculated accordingly to the height of the level to be generated.
[21] An *HashMap* is an object that works similar to a *Map*. It stores information in certain positions that are kept by *keys*. Each *key* will then contain values.

Each one of the zones mentioned in Section 4.3 are considered subclasses of the *Element* object, as exemplified in Figure 5.1's *elements* package. The class *Straight* extends the class *Element*, and calls on its own constructor the constructor of the superclass

5.2.3 Element content generation

In the previous section (Section 5.2.2), the *Cell* and *Element* objects were said to contain all of the information regarding that level's portion. Each *Element* of the level will contain a certain kind of *CellContent* that is generated exclusively for that very same *Element*. The process of generating content is sequential, and iterates through the *list* previously created for the level (refer to Section 5.2.2) generating content accordingly in each *Cell*.

```
1  begin generate
2    while i < levelCells.length
3      c := levelCells[i]
4      if i == 0 or i == levelCells.length - 1
5        c := consctructCell(CellContent.STRAIGHT)
6      else
7        if(previousContent == CellContent.WALL_JUMP or previousContent == CellContent
             .DUNKING or previousContent == CellContent.MANOEUVRE)
8          local := cellCoherence(true)
9        else
10         local := cellCoherence(false)
11       end if
12
13       c := consctructCell(local)
14     end if
15   i := i + 1
16   end while
17 end generate
18
19 begin cellCoherence
20   while cellContent == previousContent or cellContent = CellContent.WALL_JUMP
21     cellContent := pick a random content from the available content
22   end while
23 end cellCoherence
```

Listing 5.2: Methods used to generate levels in CustomizedLevel.

According to Listing 5.2, the *list levelCells* is iterated from start to end, enabling content to be generated according to the following rules:

- In case the first or the last elements of the *list* are selected, the generator will place a *Straight* zone into that *Cell*, so that starting and ending zones can be created without the presence of major obstacles. Furthermore, the existence of these zones allows the player to perceive the finality of the level;

- For all the other elements, rules are applied accordingly:
  - Despite the content of the previous *Element*, the content of the current one cannot be the same;
  - Whether the content of the previous *Element* consists of *Wall Jump*, *Dunking* or *Manoeuvre*, the content of the current cannot consist of a *Manoeuvre* zone (refer to Listing 5.2);
- As soon as the content is decided, that very same content is generated unto the *Cell* by creating the correspondent *Element* subclass.

5.2.4 Level selection

The previous chapter mentioned the techniques to create single levels, and filling in their level portions. However, in order to create unique and entertaining game levels, there is the need to perceive which ones will fit the likings of the players the most. For that effect, a fitness method has been developed, following the results of the first *social experiment* (refer to Section 6.1 for further details on said experiment). This fitness method allows the generator to understand how good the level is in terms of *fun*. Each level will have a *score* according to the level *zones* existent in the level itself. Listing 5.3 will help perceive the punctuation for each one of the *zones*, in the following format:

zoneName(zoneIndex, zoneScore)

```
1  STRAIGHT(1, 1),
2  JUMP(2, 6),
3  HILL(3, 3),
4  DUNKING(4, 7),
5  WALL_JUMP(5, 4),
6  MANOEUVRE(6, 6);
```

Listing 5.3: Code consisted in the CellContent class.

With each level, the punctuation is calculated according to the *sum* of all the *scores* of each zone consisting in that level. The final punctuation of each level is utilized to perceive which level will have the greatest points. To understand which is the level with the highest *score*, a tournament-like *competition* is held between every single level generator has created, denominated *Tournament Selection*, which is a *genetic algorithm* fitness approach.[BT95] A *list* is created with a defined number[22] of empty slots, in order to insert as many generated levels. Additionally the same number of arrays is created in order to have as many tournaments for a greater level diversity. In each tournament:

- The process is repeated until only two of the levels are kept. These two levels will then use the *crossover* technique (more details in Section 5.2.5) to create a single level.

[22] Generally, 128 (one hundred and twenty-eight).

- For each two levels, their *score* is calculated and compared. The one with the greatest *score* is preserved to the next round;

The last level will then be used in another tournament that only crossed over levels can participate on. The participants of this tournament are the resulting levels of other basic tournaments[23]. This final one will then realize which level has the greatest *score* and will then be utilized for the final generation. Listing 5.4 and Figure 5.2 show further how all the tournaments are set.

```
1  maxLevels := 128;
2  tournaments := 128;
3
4  begin createMultipleLevels
5    while i < tournaments
6      tournament := create an array of arrays for all of the tournaments
7      while j < tournament.length
8        cell := tournament[j]
9        generate(cell)
10       j := j + 1
11     end while
12
13     tournament := only the two best levels will be present
14     levels.add(crossover(tournament[0], tournament[1]))
15   end while
16 end createMultipleLevels
```

Listing 5.4: Tournament Selection method utilized.

Figure 5.2: Tournament-based decision of the generator.

5.2.5 Crossover technique

To create new and entertaining content, the generator recurs to the level *crossover* technique. It consists of fusing two different *zones* into one, by checking their *type* and adjusting their characteristics to create new *zones*. This verification is done upon joining two levels, where all the consistent *zones* will be fused according to their coordinate in the level's *list* of *Cells*. When it comes to the first and last *zones* of the level, these will remain untouched since they're both the same kind of *zone* (*Straight*). However, each different combination of *zones* creates new content. These combinations are all the possible fusions between the existent *zones* (*Straight*, *Jump*, *Hill*,

[23]The total number of tournaments is the same as the number of starting tournaments.

Dunking, *Manoeuvre* and *Wall Jump*), consisting of (number) total combinations explained in the following sections.

5.2.5.1 Organization

The classes which perform the element combinations are all based upon the *Command Design Pattern*.[GHJV94] In this case, the *Command* interface is consisted by two methods:

- **cross** - Applies all of the methods needed for each of the combinations;
- **checkElement** - Allows the generator to understand what kind of *Zone* has been created upon combination of elements.

Furthermore, to aid on the organization of the *crossover* technique, the *Crossover* object has been created. This object will allow an easier procedural call on combination methods. This object is utilized on the class *ContentPair*, which defines the kind of combination to perform and under what circumstances. According to Listing 5.5, the *ContentPair*s are created upon level generation, and assigned with the corresponding *crossover* command, to then later be executed.

```
1  /*
2  * Gets all the possible combinations for each one of the zones.
3  * Example: case JUMP: in.setCrossCommand(new StraightJump(height));
4  */
5  contentPairs = Utils.getPairCombinations(availableContent, this.height);
6
7  /*
8  * Cross command then gets called according to the stored command,
9  * allowing the creation of the new element.
10 */
11 private Element setCrossoverBlock(ContentPair contentPair, Element first, Element
       second) {
12   return contentPairs.get(contentPairs.indexOf(contentPair)).getCrossCommand().
         cross(first, second);
13 }
```

Listing 5.5: ContentPair and Crossover commands.

The following subsections will detail how different level elements can be combined.

5.2.5.2 Straight and Jump

Straight and *Jump* zones are definitely simple ones. In order to combine them, the generator takes the two elements and verifies whether the *Jump* element is the highest in floor height. In case that happens, the *Jump* element is the result of the whole operation. Else, the element will get *Ground* blocks added to the gap, starting from one square lower then the current height until the bottom of the screen (refer to figure 5.3), thus creating resonance on the ground.

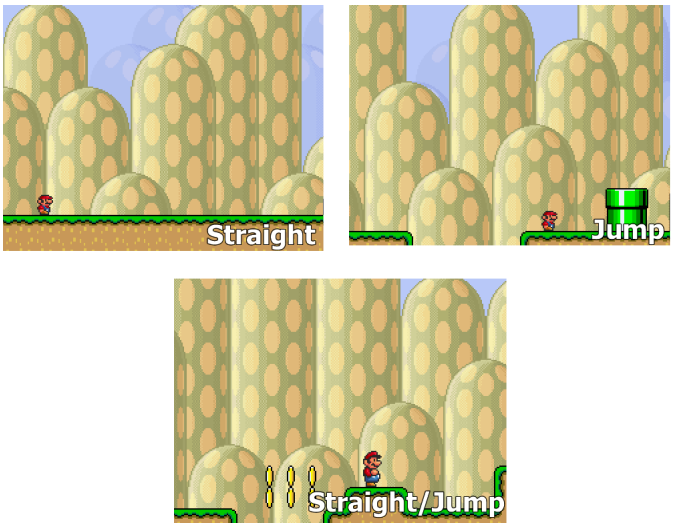

Figure 5.3: Example of a Straight/Jump fusion zone.

5.2.5.3 Straight and Hill

Each *Hill* zone can have up to *three* different hills with different heights and positions. However, when fused with *Straight* zones, these can become unique, as shown in Figure 5.4. Similar to *StraightJump*, in case the element with the *Hill* zone has the highest floor height, then the result is simply the selected element. Else, the element will be changed accordingly:

- If the *Hill* element is consistent of only one hill, then that hill is removed, and bumps are created on the floor. These bumps are created from the starting X coordinate of the hill, until then ending one, generating a random number between the floor's height and the hill's height. As soon as this number is generated, *Ground* blocks are placed accordingly upon the current X coordinate. The same operation can happen sideways, meaning that instead of creating *Ground* blocks, blocks are removed from the element's ground in order to create bumps into the landscape.

- Else, the *Hill* element will have two or more hills. The hills from this element are copied into the *Straight* element, creating then a *Hill* zone with the floor height of the *Straight* one.

5.2.5.4 Straight and Dunking

Dunking is a zone that is completely related to the *Straight* one, since the only addition to it are enemies. Regardless, the two zones are perfectly eligible to be fused into one. The general idea of the combination is to create bumps by fusing both of the element's heights, adding or removing blocks in key X coordinates to create said bumps.

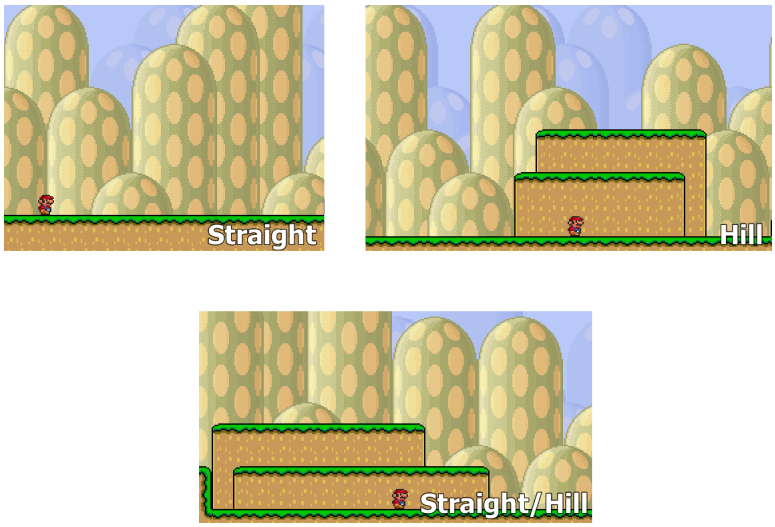

Figure 5.4: Example of a Straight/Hill fusion zone.

5.2.5.5 Straight and Manoeuvre

Despite being two completely different kinds of zones, *Straight* and *Manoeuvre* are perfectly compatible to fuse with each other. Since *Manoeuvre* is an ability zone (refer to Section 4.3.5), the combination of both should end in an area that requires some more skill from the user. With that in mind, the amount of stones from the *Manoeuvre* zone is saved, and then placed vertically, having each column leave a blank space in between both of them, as illustrated on Figure 5.5. This zone will then be another one where all players will require skill and preciseness to complete.

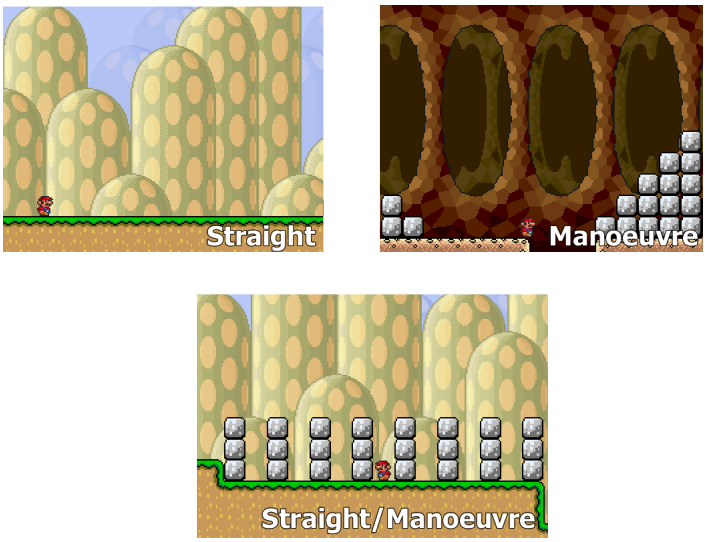

Figure 5.5: Example of a Straight/Manoeuvre fusion zone.

5.2.5.6 Straight and WallJump

The *Wall Jump* zone is yet another zone that requires skill and preciseness from the player. Since both are distinguished from each other, fusing them is not a hard task if bumps are simply created within the safe area of the *Wall Jump* element. In case this element has the highest floor, then each bump will have the probability of being as deep as the height of the *Straight* element, since the depth of the bump is randomly generated. Else, the resulting element will simply be the *Wall Jump* element. Figure 5.6 demonstrates the result with the bumps applied to the elements.

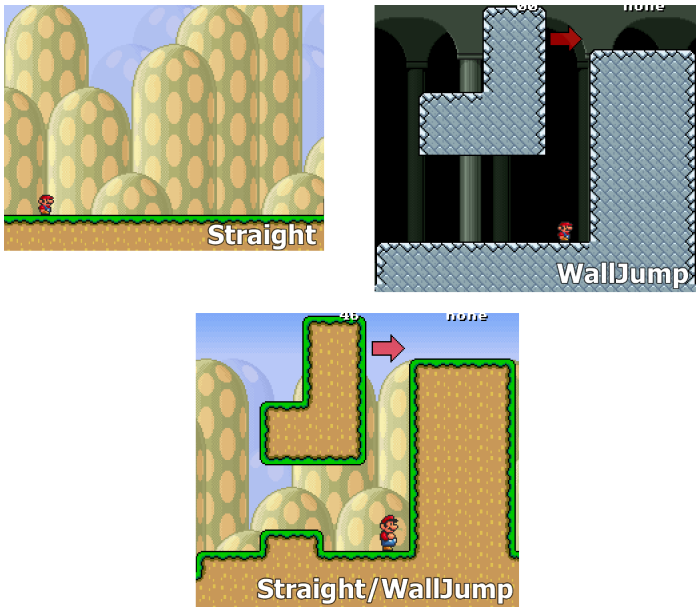

Figure 5.6: Example of a Straight/WallJump fusion zone.

5.2.5.7 Jump and Hill

To consider these two zones fused with one another is perfectly eligible. Either the *Jump* or *Hill* elements are independent ones, and their fusion will result in another brand new and original element, as simple as the resulting zone might be. The fusion of the elements will result in a normal *Jump* zone with hills attached to its background (as illustrated on Figure 5.7). The decision of which element to take is based on its floor height. Whether the *Hill* element has the lowest floor height, a gap will be created within the very same X coordinates as it is located in the *Jump* element. Else, in case the opposite happens, the *Jump* element will have the *Hill* element's hills attached to the first element. The result will always be the same, regardless of which element has a higher or lower floor height.

5.2.5.8 Jump and Dunking

Since *Dunking* is a zone that is by all means similar to the *Straight* zone, the fusion of both will result in a similar one to the *StraightJump* one (more details on Section 5.2.5.2). However, the

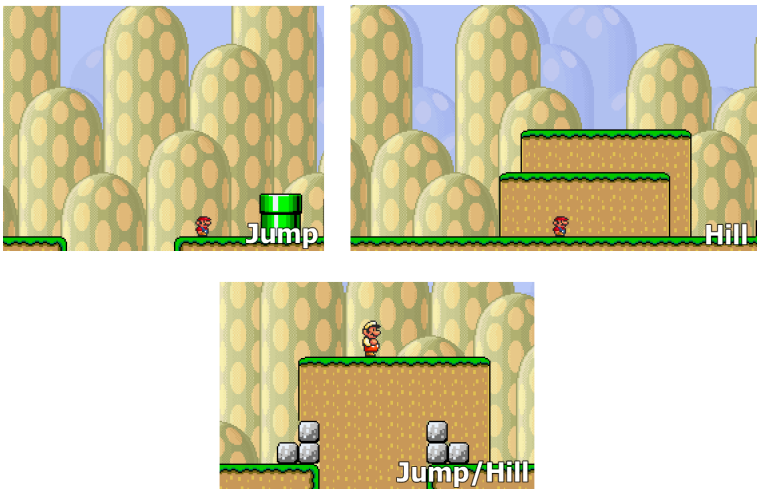

Figure 5.7: Example of a Jump/Hill fusion zone.

Dunking zone has its focus on enemies. The resulting zone will use the amount of enemies in the *Dunking* one and attach them to the *Jump* zone, in coordinates where the enemies can walk or jump on. Figure 5.3 illustrates this fusion.

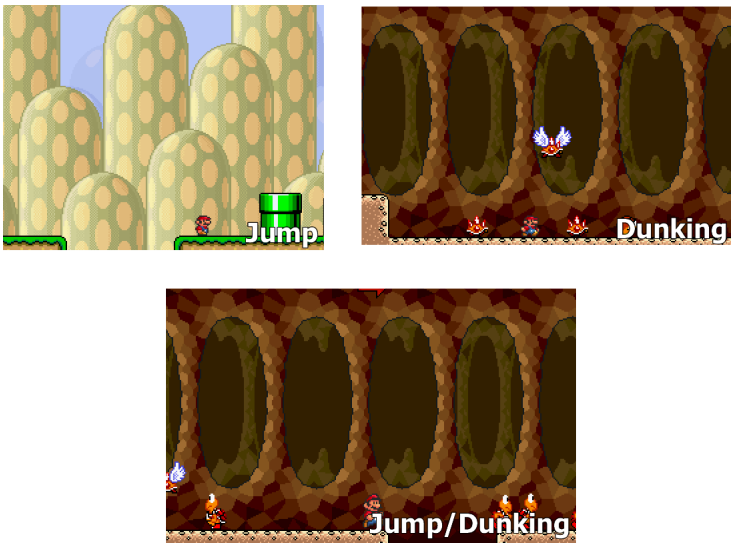

Figure 5.8: Example of a Jump/Dunking fusion zone.

5.2.5.9 Jump and Manoeuvre

As previous stated in Section 5.2.5.9, *Manoeuvre* zones tend to be areas where the players' skills will be put test. With this in mind, the *JumpManoeuvre* zone was created to ensure these skills are triggered. The result is similar to the one of the *StraightManoeuvre*, but with a gap. *Rock* blocks are placed accordingly by columns separated by one single space, until the end of the level screen. In case the player misses one of the *Rock* blocks, *Mario* falls down the gap, and dies. Figure 5.9 can illustrate these newly created zones, and the challenges they pose.

Figure 5.9: Example of a Jump/Manoeuvre fusion zone.

5.2.5.10 Jump and WallJump

Despite the unique name of this zone, *JumpWallJump* is a complex zone that will include both the *Wall Jump* and *Jump* zones' features. The *WallJump* will have an addition of a gap. The gap's properties will be the same ones of the gap included in the *Jump* element, such as start and ending coordinates. According to Figure 5.10, this will result in the element containing a gap before the *Wall Jump* obstacle is posed to the player.

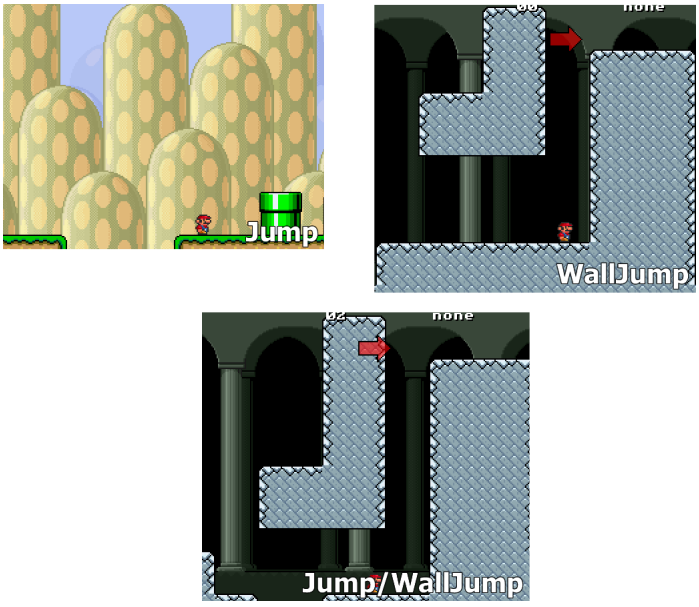

Figure 5.10: Example of a Jump/WallJump fusion zone.

5.2.5.11 Hill and Dunking

Since *Dunking* zones are related to enemy spawning zones, the general idea of fusing these with *Hill* zones is simply achieved by the generator. The idea is to place as many enemies as the

Dunking allows on top of the hills the other zone contains. The result will be quite challenging, as the enemies are all replaced by *Red Koopas*, enemies that do not fall from ledges. This way, the hills will be populated as long as the player does not eliminate the enemy threats.

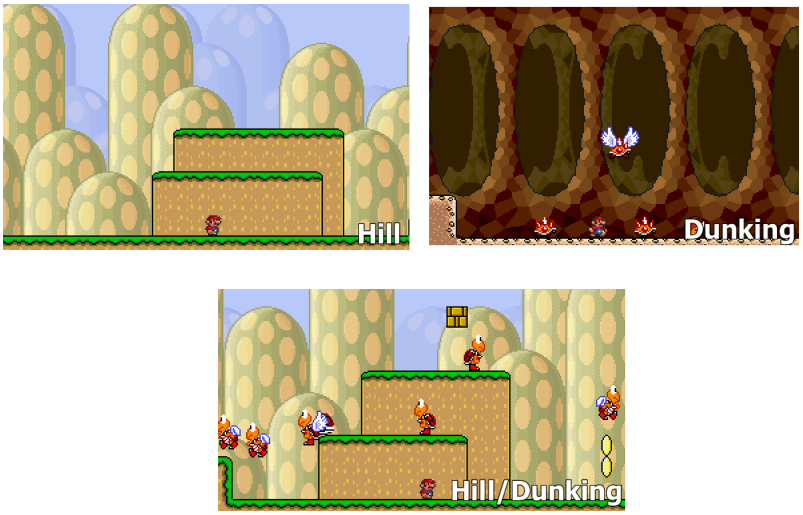

Figure 5.11: Example of a Hill/Dunking fusion zone.

5.2.5.12 Hill and Manoeuvre

These zones are formed by the simple fusion of both elements. In other words, *Manoeuvre* zones get the hills the *Hill* element contains added to its background, creating a zone with higher risk of death, and higher difficulty to eliminate enemies that are found around this zone.

5.2.5.13 Hill and WallJump

As the aim of the project is to create general and entertaining content for players to be feel entertained in, this zone is a special fusion between the *Hill* and *WallJump* elements. As placing simple hills on the background layer of a *WallJump* zone would render the *Wall Jump* technique useless (due to the hill's placement, the player could have the opportunity to simply jump over the ledges of the hill and ignore the walls), a while new area was needed. This fusion aimed to create content which would allow players to use the *Wall Jump* technique with the existence of hills on said element. For this effect, Figure 5.12 illustrates the scenario. The needed walls composing the *WallJump* element were placed on the very edges of the zone, and hills were created in between, so players could use the technique to reach the hills and then advance throughout the level.

5.2.5.14 Dunking and Manoeuvre

In order to create challenging areas, *Rock* blocks from *Manoeuvre* zones have been added to *Dunking* zones in an organized manner, to emphasize the concept of *Dunking*[24]. According to

[24]Throw objects from a higher place in order to cause damage to said objects, or others located under the shooter.

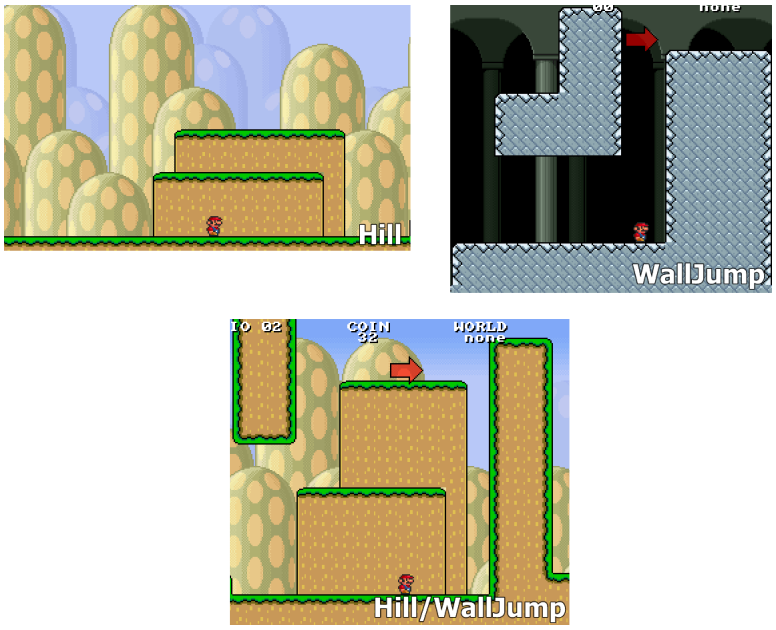

Figure 5.12: Example of a Hill/WallJump fusion zone.

Figure 5.13, blocks have been added to both sides of the zone, to *trap* enemies, as other blocks were added to airborne locations in order for *Mario* to stand and shoot *Fireballs* (in case *Mario* is currently on the *Fire* state).

Figure 5.13: Example of a Dunking/Manoeuvre fusion zone.

5.2.5.15 Dunking and WallJump

The fusion at hand consists of enemies from a *Dunking* element on a *WallJump* element. In all basis, the *WallJump* element will be consisted of enemies, making the player's path harder to cross and complete (as illustrated on Figure 5.14).

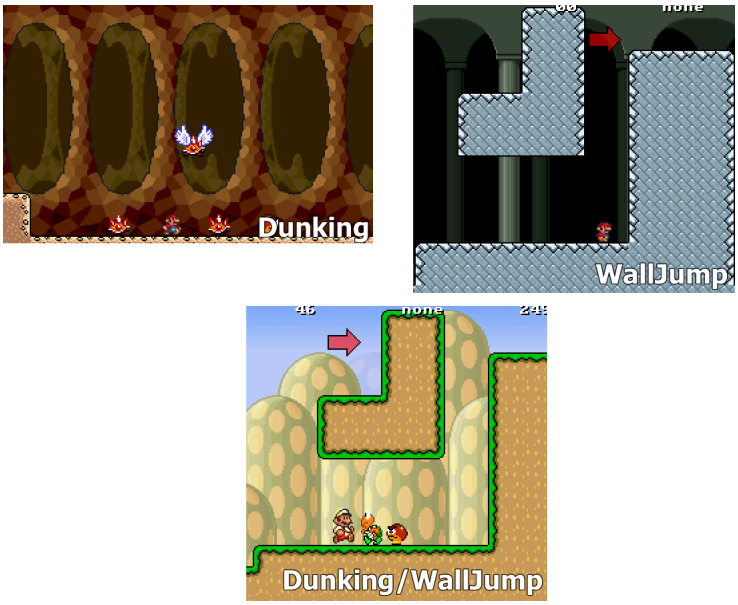

Figure 5.14: Example of a Dunking/WallJump fusion zone.

5.2.5.16 Manoeuvre and WallJump

Based on the *WallJump* zone, this fused one is a simple adaptation of the first. As the *rock* blocks are important elements of the *Manoeuvre* zone, these have been used to benefit the newly created one, by augmenting the difficulty of the scenario posed to the players. *Rock* blocks are added the walls, in groups of two, consisting of a block gap distancing the groups. The player's *Wall Jump* process will be complicated, thus challenging the player's skills. Figure 5.15 can be referred to for this zone's layout.

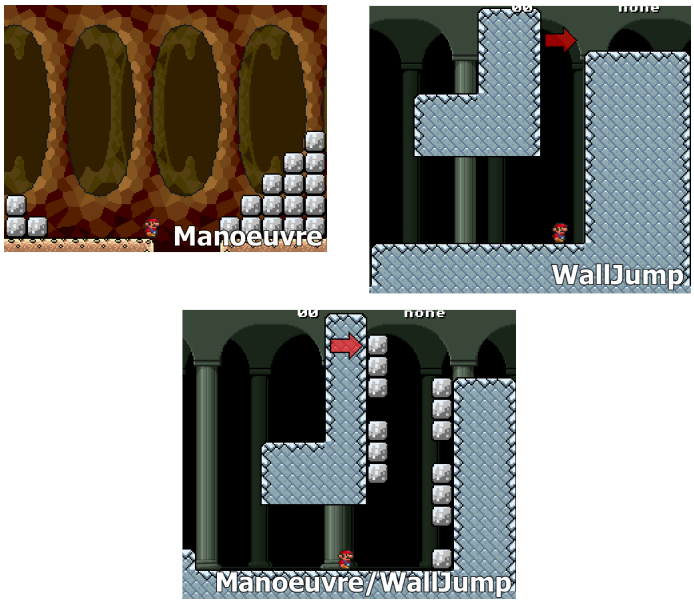

Figure 5.15: Example of a Manoeuvre/WallJump fusion zone.

5.2.6 Final assembling

Selecting what zones the level will be consisted of is considered a task of great importance. However, adding extra assets to the level boosts the visual content, and opens new gates to the player's experiences. In this chapter, the addition of assets and how they influence gameplay will be analysed.

5.2.6.1 Probability calculation

Each one of the level's extras (coins, blocks and enemies) will have their own probabilities. During parameter selection, each probability is defined as a percentage, ranging from 0 to 100 as unit[25] values. These numbers are generated, in unit values, from 0 to the value in range minus 1. As for the calculation of which number to place in the generation, the percentage number is taken and used to divide the number 100 (example: In case the percentage is 50, the resulting number should be 2 through the calculation *100/50*). Each resulting probability will be utilized in the *Extras* class, explained in the next section.

AS for the generation of numbers, since the *Random* object allows the creation of random numbers throughout the mechanism of *seeds*[26], a method was created to get numbers according to a normal distribution (refer to Listing 5.6). This kind of distribution was used so that the normal sequence of numbers within the *Random* object would be avoided, therefore generating numbers in a more effective manner.

```
1  begin randomDistrib(maxNumber)
2    while r > 1 or r == 0
3      x = 2.0 * randomDouble - 1.0;
4      y = 2.0 * randomDouble - 1.0;
5      r = x*x + y*y
6    end
7
8    finalResult = floor(r*maxNumber)
9    return finalResult
10 end randomDistrib
```

Listing 5.6: Normal number generation method.

The calculation utilized in Listing 5.6 is based on the *Monte Carlo Method* for random number generation within a normal distribution. [Hro10] The calculation requires an inicial variable (in thise case, called *r*) to be placed within the number 0 and 1, throughout the generation of two other variables (*x* and emphy). Applying the calculation in Listing 5.6's line 5, we obtain the initial variable. Finally, multiplying the resulting variable by the maximum number allowed for the generation will result in the new-found random number.

[25] One by one.

[26] A *seed* allows the generator to know where to get the random number generation should start getting the numbers from.

5.2.6.2 Extras class

In order to aid in the creation of extra content (such as blocks and coints), the *Extras* class becomes useful. This class is represented by methods that will allow the creation of extra content within the level, such as coins, blocks and enemies. The *Extras* class contains all of the methods needed to endorse the level.

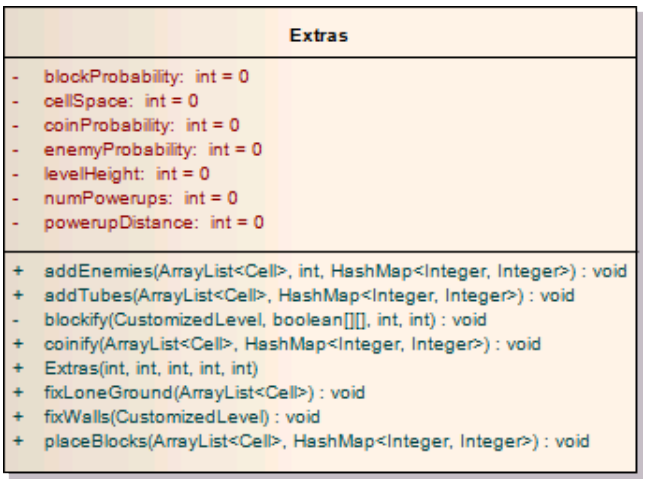

Figure 5.16: The Extras class's methods.

- **coinify** - This is the method that will be used to create coins on each one of the zones of the previously created level. Coins are placed according to the probability defined by the user upon selecting the game parameters (more details on Section 4.1), and according to the height of the floor they should be placed on. Each coin placement follows the restrictions posed in the *CellConstraints* interface;

- **placeBlocks** - Similar to the *coinify* method, *placeBlocks* will have the job of placing item blocks throughout the zones of the generated level. These blocks also follow all of the constraints previously posed;

- **addEnemies** - Yet another element that is controlled through the probabilities posed by players, *enemies* are extra features that endorse the game scenario. These enemies are added according to the zones they will be inserted in. This method does not allow enemies to be placed in safe zones, such as the starting and ending ones. Furthermore, to understand which enemies to generate, the program takes into account certain byte values. Each one of these values represents an enemy to generate, as presented in the Listing 5.7.

- **addTubes** - As the name implies, this method will add tubes to the generated zones. Every single zone will have the opportunity to contain tubes, except for the *Dunking*, *Manoeuvre* and *WallJump* elements. All of the other zones can have tubes, which will endorse the level's graphic quaility. Each tube occupies two squares in length, and a maximum of two squares of height (imposed by *CellConstraints*);

- **fixLoneGround** - Method that goes through the level and analyses *Ground* blocks that are standing alone in some areas. These blocks are then erased in order to correct the level visual aspect;
- **fixWalls** and *blockify* - Methods which create the level walls and barriers, so that *Mario* can step platforms safely.

```
1 ENEMY_RED_KOOPA = 0;
2 ENEMY_GREEN_KOOPA = 1;
3 ENEMY_GOOMBA = 2;
4 ENEMY_SPIKY = 3;
5 ENEMY_FLOWER = 4;
```

Listing 5.7: Byte values for the level enemies.

All of these methods are then called upon the final generation of assets into the level, as soon as all of the crossover techniques have been applied.

5.3 Chapter Summary

The current chapter presents the implementation techniques utilized throughout the implementation of the level generator. Applying constraints to the level itself, the generator can ensure a higher quality in the levels generated and its portions. Each one of the portions is then created and content is consequently generated in each of them, throughout the randomization and the selection of pre-defined zones, described in Section 4.3.

Finally, as these levels are created, a selection of the best levels is of importance, to ensure maximum entertainment to the user who wishes to generate levels. For this effect, the generator applies a *genetic algorithm technique* in which it performs a selection of the best *individuals* (being a level an individual) throughout a tournament-like selection (refer to Section 5.2.4). This tournament will then decide which ones are the best levels, to then apply a *crossover* technique on both of them, to generate new content.

As the final level is finally created, the insertion of extra content (such as *blocks* or *coins*) is executed, creating a far more pleasant level, as well as challenging due to the amount of obstacles added.

Chapter 6

Experiments & Results

In order to address the data needed for the generator to create levels and eventually its validation, two *social experiments* were performed. These experiments counted with, in total, around *thirty* participants, enhancing largely the information needed for the generator to perform its actions. The current chapter aims to provide all of the information retrieved from these experiments, as well as the procedures taken.

6.1 Super Mario Bros. Experiment

Being the first of two experiments, the *Super Mario Bros. Experiment* had the purpose of retrieving data from existing levels on the *Mario* universe added to the participants' opinions about them. With the participants' inputs, pertinent data was given the opportunity to be organized and used in the generator to create entertaining content for players and to find a definition of *fun* within the *Mario* universe. Since great part of the participants were players themselves, the input became even more valuable.

6.1.1 Experiment procedure

The experiment is based on playing the popular game *Super Mario Bros.* from the *Nintendo Entertainment System*, similar to the one used to generate levels. Henceforth, participants were asked to play three existing levels in the game (refer to Figure 6.1). These levels are different in difficulty and in the content they present to players:

- *Level 1-1* - The very first level of the game. This is a simple level, where the difficulty is deemed as easy. Consisting solely of areas where the player just runs straight, and of some gaps and tubes, the level is simple and entertaining;

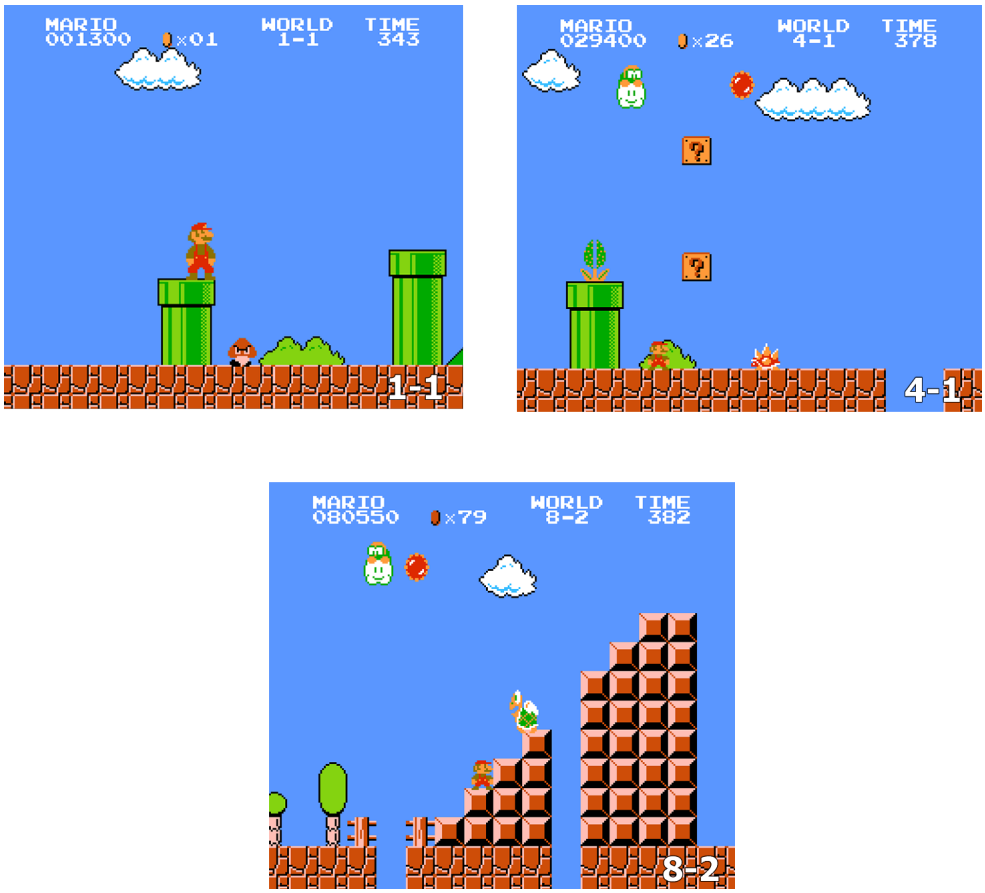

Figure 6.1: *Super Mario Bros.*'s levels 1-1, 4-1 and 8-2, respectively.

- *Level 4-1* - Augmenting the difficulty in relation to the first level, level 4-1 presents to its players some obstacles, requiring the players' skills, despite being a *fast* level. These skills are set to up to an average difficulty;

- *Level 8-2* - As the *Super Mario Bros.* game is consisted of 8 worlds (refer to Section 2.2), the eighth one presents the players with levels which difficulty has been raised to a higher point. By including zones where the player must manoeuvre his skills, this level poses a greater challenge to players, thus enhancing the entertainment rate.

However, in order to play the game, an emulator was needed. The used *emulator* was *BizHawk*[27], a program with capabilities of running *Nintendo Entertainment System* games and record their gameplay. This emulator was chosen so that records of the players could be attained for further research on level properties. Each participant would play the required levels once, and therefore have their performance recorded. At the end of each of participant's contribute, they would be requested to fill in a survey according to the experience they were apart of.

[27] https://code.google.com/p/bizhawk/

6.1.2 Survey presented

Each one of the questions presented on the survey are related to the participants' feelings towards the recorded gameplay, or the levels at hand. The objective of the survey consisted in trying to understand what parts of the levels were the most interesting and the general feeling on the game itself, as shown in Appendix A.

- **Question A.1** - Several generations have played the *Super Mario Bros.* games, and hold several different opinions on them. To understand the likings of each generation, this question was posed.
- **Question A.2** - As the level generator will have to understand what the most entertaining features that a certain player needs in order to find the game fun, this questions aids in discovering what the participants' preferences are.
- **Question A.3** - Following the guidelines of the previous question and to establish a concurrent connection between *fun* and *challenge*.
- **Question A.4** - For each of the levels played (1-1, 4-1 and 8-2), each participant was questioned on how amusing or challenging the experience was. These questions had, as an answer, the option of picking a number in a scale of *one* to *five*, where *one* would refer to "*Not at all*" and *five* to "*A lot*".
- **Question A.5** - This question was posed in order to understand what would be player's expectations in relating to the difficulty found in the generated levels. In on hand, players could find easy levels quite amusing, but not challenging at all or, on the other hand, hard levels could be challenging but not *fun* to play.
- **Question A.6** - A pack of *four* questions were posed to participants to understand which elements of the level should be more present in the final outcome of the generation. The *X* marked in the question would be replaced by one of the following, leaving the participant the choice of the answer within a scale of *one* to *five*, similarly to the Question A.4.

6.1.3 Survey results and conclusions

As it can be observed from Figure 6.2 and Table A.1, the majority of the participants were under the age of *thirty*. Majority of the youngest population of the world plays video games, making their opinion a lot more insightful. [Gee05]

Regarding the elements that constitute a level, majority of participants rated *Enemy* zones (known as *Dunking* zones) as the most amusing on a level, whereas *Manoeuvre* zones were deemed the most challenging. The results presented in Figure 6.3 and Table A.2 allowed the calculations for the points settled for each one of the zones within the generator (refer to Section 4.3). These results were calculated according to several formulas, starting with Equation 6.1.

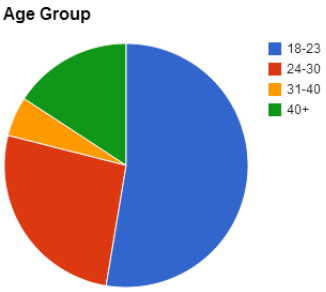

Figure 6.2: Question A.1's graphical results.

$$\frac{Surveyscore}{Total participants} = \frac{Result}{Maximum points (scale)} \tag{6.1}$$

- *Survey score* - The number of votes for that zone type;
- *Total participants* - The total number of survey participants;
- *Result* - Desired calculation result;
- *Maximum points (scale)* - The maximum allowance for points per zone, which will always depend on the scale allowed. *Ten* (10) was the allowed scale in this case.

Equation 6.1 was utilized to achieve results for each one of the zones and therefore be introduced into the final calculation. The final values will always reflect the *fun* and *challenge* factors. Since both factors are taken into account when generating a level, the final calculation would be an *average*[28] between both *fun* and *challenge* results for each one of the zones. To add to the average, the values in Table A.5 are also taken into account:

Figure 6.3: Questions A.2 and A.3's graphical results.

- Question **6.1** - According to the results, participants believe that the quantity of gaps should be balanced;

[28] A simple calculation that involves the sum of all of the needed elements, to then be divided by said number of elements. This allows to calculate tendencies in large scale groups. [HTMA82]

- Question **6.2** - As for hills, the opinion remains the same;
- Question **6.3** - However, enemies should have their numbers increased according to the participants' opinions;
- Question **6.4** - Regarding blocks, their opinion states these should be in a balanced number.

For each of the results previously stated, Table 6.1 was created to ensure the perfect score for each of the zones.

Zone	Fun	Challenge	Quantity input	Final Result
Straight	1	0	-	1
Jump	3	9	6	6
Hills	1	0	5	3
Enemies (Dunking)	8	6	8	7
Blocks (Wall Jump)	5	3	5	4
Manoeuvre	4	7	-	6

Table 6.1: Final zone punctuation.

Each level has its characteristics, being these characteristics what makes the level harder or more amusing. These questions reflect the participants' opinions relating the *fun* and *challenge* factors present in the played levels (refer to Table A.3).

- *Questions **4.1** and **4.2*** - These questions pose results towards Level 1-1, the easy one. Results (refer to Figure 6.4) show that the level is not *challenging* according to majority of participants, but it remains *fun* in its core.

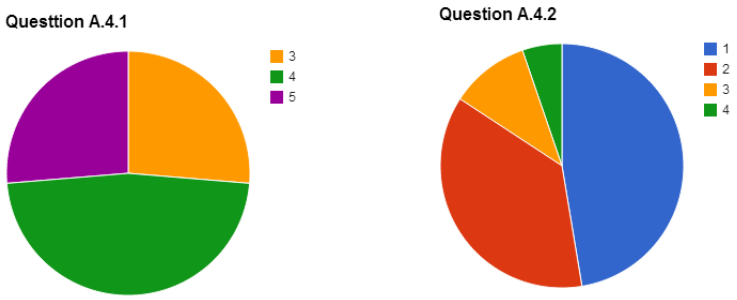

Figure 6.4: Questions **4.1** and **4.2**'s graphical results.

- *Questions **4.3** and **4.4*** - Relating to Level 4-1, participants show that the *fun* factor raises compared to the first level, as well as the *challenge* rate, according to Figure 6.5.
- *Questions **4.5** and **4.6*** - As for the final level (Level 8-2), participants' opinions fluctuate. Some of the players were not able to complete the level successfully, therefore having their opinion on the *fun* factor influenced negatively. However, those who did complete the level found it extremely *fun*. Both groups were able to concede that this last level was the most challenging, according to Figure 6.6.

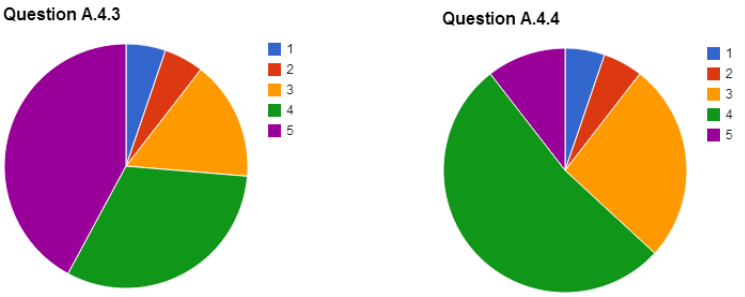

Figure 6.5: Questions **4.3** and **4.4**'s graphical results.

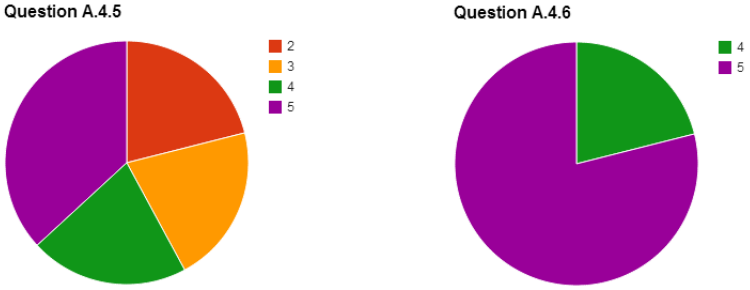

Figure 6.6: Questions **4.5** and **4.6**'s graphical results.

Still connected to the group of Question A.4, Question A.5 (refer to Table A.4) shows that difficulty can be related to the level of *fun* and *challenge* each one of the generated worlds has to offer. In this case, the higher the difficulty, the higher *fun* and *challenge* became.

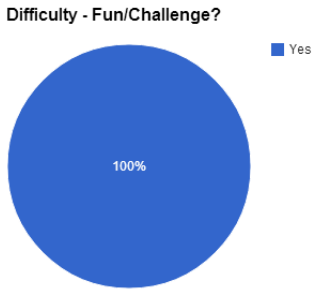

Figure 6.7: Question A.5's graphical results.

6.2 Super Mario Bros. Generated Experiment

Upon completion of the level generator previously proposed, the need of validation is at hand. This experiment aims to understand whether the generator works as it should, and if all of the generated levels could be easily mistaken as normally designed levels. The objective was to have the very same population of participants as in the first experiment, so the results could have a lot

more impact. Turns out that most of this experiments' participants were also players of the original *Super Mario Bros.* game.

6.2.1 Experiment procedure

In order to have each participants' input, *three* levels were generated according to the parameters described in Table 6.2.

	First Level	Second Level	Third Level
Level kind	Normal	Normal	Normal
Level length	180	255	390
Level type	Underground	Overground	Castle
Difficulty	Easy	Normal	Hard
Time limit	300	450	900
Coin probability	50	50	50
Block probability	50	50	50
Enemy probability	20	30	35
Power-up distance	2	2	2
Mario lives	3	5	7
Wall Jump limit	-	-	5
Seed	No seed	No seed	No seed

Table 6.2: Second Experiment level parameters.

At the end of the experiment, participants were asked to answer a simple survey, described on the next section.

6.2.2 Survey presented

Each one of the questions presented on the survey in Appendix B are related to the participants' experimented levels. The objective of the survey consisted in trying to understand whether the generator was completely successful at providing entertaining, challenging and original levels.

1. **Question B.1** - Similarly to the first experiment, it became important to retrieve the age group the participants were inserted in, so that it is ensured that various generations of players would be satisfied or not.

2. **Question B.2** - For each of the three levels played, each participant was questioned on how amusing or challenging the experience was. These questions had, as an answer, the option of picking a number in a scale of *one* to *five*, where *one* would refer to "*Not at all*" and *five* to "*A lot*".

3. **Question B.3** - In order to understand whether the results of the first experiment will match the same ones after a generated experience, this question was posed again.

4. **Question B.4** - Similar to the first experiment, and following the guidelines of the previous question and to establish a concurrent connection between *fun* and *challenge*.

5. **Question B.4** - For each one of the played levels, players had an opinion regarding their originality. This question aimed in assisting the clarification of whether these levels were indeed generated, or would blend in with manually created levels.

6.2.3 Survey results and conclusions

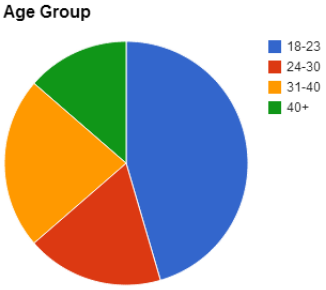

Figure 6.8: Question B.1's graphical results.

As of Table B.1 and Figure 6.8, we can depict that the population of the participants' age group is similar to the one of the first experiment (check Table A.1).

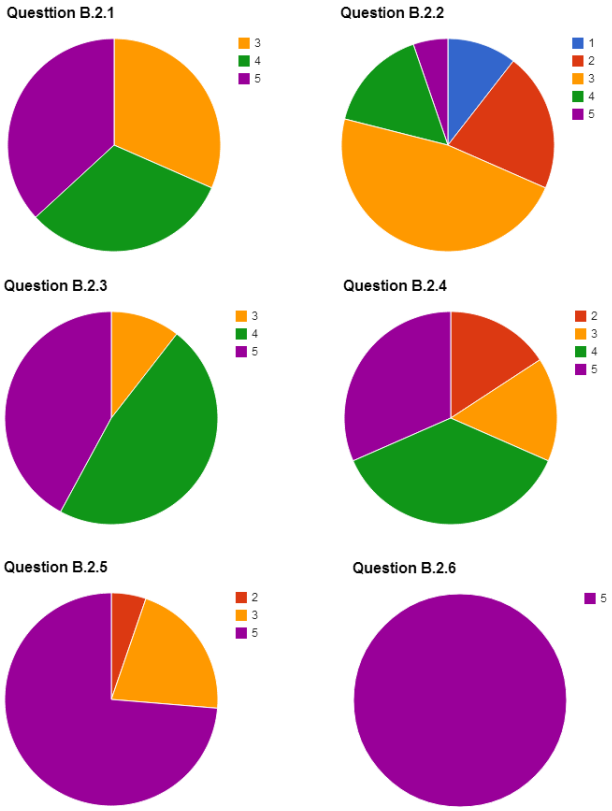

Figure 6.9: Question B.2's graphical results.

Similarly to the previous experiment, each level has its characteristics, making the level harder or more amusing. The answers presented in Table B.2 and Figure 6.9 reflect the participants' inputs on each one of the levels. From the results, we can depict that the generator became successful in its task of providing fun to the participants. As expected, the easy level will have a good balance between the *fun* and *challenge* factors, whereas the hard level will maintain both in an high-valued stance. As for the normal level, the level presented general *fun* for the audience, revealing itself *challenging* in certain aspects.

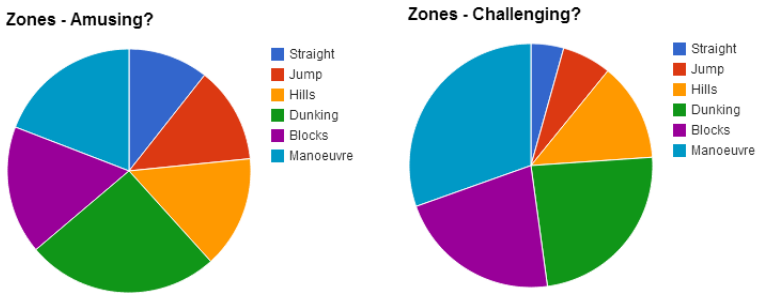

Figure 6.10: Question B.3 and B.4's graphical results.

As presented on Table B.3 and Figure 6.10, each one of the level zones has its impact on the generator. In general, the audience believes that each one of the zones can be *fun*, but constricts the *challenging* factor to zones which require far more skill from the players.

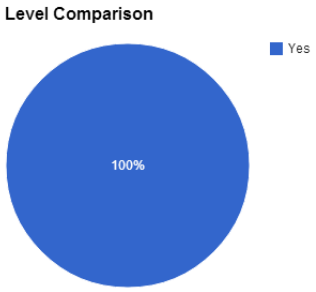

Figure 6.11: Question B.5's graphical results.

Generally speaking, the experiment revealed itself a success, where most of the participants even stated that the generated levels looked like manually designed *Super Mario Bros.* levels, since most of them, as previously stated, were players of the original *Super Mario Bros.* game, and also participants on the first experiment. According to Table B.4 and Figure 6.11, it can be concluded that this last statement boosted the level generator's validity further.

6.3 Chapter Summary

In this chapter, two surveys and their corresponding results have been presented in order to back up the *social experiments* performed. These experiments aimed to understand the vital information in order to develop a level generator, as well as validate said generator.

The first experiment had the purpose of retrieving information relating to the *Super Mario Bros.* game, and which would be the best aspects to implement on a level generator. Several participants stated their opinion, and these results were then utilized to develop the entertaining level generator.

However, the second experiment based itself in trying to validate both the level generator and the hypothesis explained in Section 1.2. The results turned out to be positive, and the participants believe that the generator does create authentic *Mario* levels.

Chapter 7

Conclusion

This chapter presents information regarding objective completion and future research questions. Furthermore, there is the need for an analysis of the previously stated objectives, and whether these were accomplished.

7.1 Objectives accomplished

According to Section 1.2, the objectives of this dissertation project refer themselves to the creation of a level generator capable of creating entertaining and original levels, objective that by itself became fulfilled.

- *Fabricate a procedural level generator* - In Chapter 6 we show that there is some empirical evidence that the generated levels are indeed fun to both casual and experienced players.;
- *Unique and immerse game mechanics* - Needed game mechanics were enhanced ever since the addition of *Dunking* and *Wall Jump* zones, zones that by themselves require a lot more effort of the player and consequent augment of his skills, as well as the level of immersion presented;
- *Nested artificial intelligence* - Genetic algorithms (tournament decision algorithms, expanded further in Section 5.2.4) were applied in order to constitute several levels, only with the best information and generated content.

Based in the initial hypothesis, and after all of the research, experimentation and validation, it is possible to create fun and challenging levels, according to the crossover techniques (detailed in Section 5.2.5) and the genetic algorithms utilized (tournament selection, detailed in Section 5.2.4). Additionally, the experiments conducted with players revealed themselves to confirm the clarity of the solutions presented to answer the initial hypothesis: the generated levels were indeed deemed as challenging and fun (refer to Section 6.2).

7.2 Fun vs Challenge

One of the main focuses of the dissertation presented was the comparison and contrast of the *fun* and *challenge* concepts. Both these concepts are connected with each other, as we can conclude from the second *social experiment* (refer to Section 6.2). Regardless of depending on players' opinions' subjectivity, it is highly recommended that a correlation of both is present on the generated levels. Fun can be *found* in a game through its gameplay or through the features presented to the player. *Simplicity* can take a great part of a player's fun *measurement*, as well as the amount of content presented by a single feature, therefore making content challenging. [Kos13]

Challenge however, presents a higher form of entertainment for certain users. According to the first experiment, certain players become attracted to more challenging levels than others. There are players that rather depict easy levels (not challenging at all) far more fun, due to the power that it gives to the player. The player will feel omnipotent, and completely invincible in case the difficulty is easy, and some players find *fun* within these cases. The term *challenge* had a complex and subjective meaning, since it varies from person to person.

7.3 Future Work

A possible addition to the current project would be the creation of several artificial intelligence agents that could play the generated levels at any cost. These agents could be subdivided into several categories, having their capacity led by different player mentalities, such as *time attacking*, *exploring* or even *novice* AIs that will behave like inexperienced players.

Regarding the level generator, it can be concluded that innovation can always be boosted. One of the ideas that came to mind during the project's development lifetime was the creation of *vertical* levels. This can be achieved just by implementing vertical zones, by considering the Y coordinates instead of focusing on the X ones. However, before advancing into such idea, some of the current zones can still be improved, which hasn't been done due to the lack of sufficient time. Another possible research and eventual implementation could be the application of the techniques utilized in this dissertation project to other platform games.

Additionally, a real *Turing* test can be performed on the generator. According to Question B.5's answers, most of the participants of the second experiment (refer to Section 6.2) claimed that the levels were similar to original *Mario* levels. However, these participants did not play a manually created *Mario* level. Just a statement is not strong enough to compel to the generator's objective, but it can be done accordingly.

References

[AK12] Kim Adelhardt and Nedyalko Kargov. Mario game solver. *IT University of Copenhagen*, 2012.

[BC10] Slawomir Bojarski and Clare Bates Congdon. REALM: A rule-based evolutionary computation agent that learns to play Mario. *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pages 83–90, August 2010.

[Bra06] David Braben. Q&A: David Braben–from Elite to today, 2006.

[BT95] Tobias Blickle and Lothar Thiele. A mathematical analysis of tournament selection. pages 9–16, 1995.

[Dou08] Andrew Doull. The Death of the Level Designer. *Procedural Content Generation Wiki*, 2008.

[FF01] Ari Feldman and Arl Feldman. *Designing arcade computer game graphics*. Wordware Pub., 2001.

[Gee05] James Paul Gee. Good video games and good learning. *University of Wisconsin-Madison*, 85(2):33, 2005.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[Gu06] Bjarki Gu. Procedural content generation. *Self-made*, 2006.

[Hro10] Juraj Hromkovic. *Algorithmics for hard problems: introduction to combinatorial optimization, randomization, approximation, and heuristics*. Springer-Verlag, second edition edition, 2010.

[HTMA82] Fumio Hayashi, Q Tobin's Marginal, and Q Average. A neoclassical interpretation. *Econometrica*, 50:731–753, 1982.

[Hug] Mark Damon Hughes. Game Design: Article 07: Roguelike Dungeon Generation.

[Kos13] Raph Koster. *Theory of fun for game design*. " O'Reilly Media, Inc.", 2013.

[KT12] Sergey Karakovskiy and Julian Togelius. The Mario AI Benchmark and Competitions. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):55–67, 2012.

[Lai04] John E Laird. History of computer games, 2004.

REFERENCES

[LB09] Bill Loguidice and Matt Barton. *Vintage games: an insider look at the history of Grand Theft Auto, Super Mario, and the most influential games of all time*. CRC Press, first edition edition, 2009.

[MdSB11] Fausto Mourato, Manuel Próspero dos Santos, and Fernando Birra. Automatic level generation for platform videogames using genetic algorithms. *Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology - ACE '11*, page 1, 2011.

[PJS04] Heinz-Otto Peitgen, Hartmut Jürgens, and Dietmar Saupe. *Chaos and fractals: new frontiers of science*. Springer, 2004.

[R. 97] Glenn R. Wichman. A Brief History of "Rogue", 1997.

[STY+11] Noor Shaker, Julian Togelius, Georgios N Yannakakis, Ben Weber, Tomoyuki Shimizu, Tomonori Hashiyama, Nathan Sorenson, Philippe Pasquier, Peter Mawhorter, Glen Takahashi, Gillian Smith, and Robin Baumgarten. The 2010 Mario AI Championship: Level Generation Track. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(4):332–347, 2011.

[STY+13] Noor Shaker, Julian Togelius, Georgios N Yannakakis, Likith Poovanna, Vinay S Ethiraj, Stefan J Johansson, Robert G Reynolds, Leonard K Heether, Tom Schumann, and Marcus Gallagher. The Turing Test Track of the 2012 Mario AI Championship : Entries and Evaluation. *IEEE Symposium on Computational Intelligence and Games, 2012. CIG 2012.*, 2013.

[SYT13] Noor Shaker, Georgios N Yannakakis, and Julian Togelius. Crowdsourcing the aesthetics of platform games. *Computational Intelligence and AI in Games, IEEE Transactions on*, 5(3):276–290, 2013.

[TKK09] Julian Togelius, Sergey Karakovskiy, and Jan Koutn. Super Mario Evolution. *IEEE Symposium on Computational Intelligence and Games, 2009. CIG 2009.*, 2009.

[TKSY11] Julian Togelius, Emil Kastbjerg, David Schedl, and Georgios N Yannakakis. What is Procedural Content Generation? Mario on the borderline. *PCGames '11 Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*, 2011.

[TSKY12] Julian Togelius, Noor Shaker, Sergey Karakovskiy, and Georgios N Yannakakis. The Mario AI Championship 2009 – 2012. *IEEE Symposium on Computational Intelligence and Games, 2012. CIG 2012.*, pages 1–9, 2012.

Appendix A

First Survey

A.1 Age group

- 18-23;
- 24-30;
- 31-40;
- 40+;

A.2 What do you consider fun in platform games?

- Straight;
- Jumping gaps;
- Hills;
- Enemies;
- Blocks (obstacles, power-up blocks, etc.);
- Manoeuvre zones;

A.3 What do you consider challenging in platform games?

- Straight;
- Jumping gaps;
- Hills;
- Enemies;

- Blocks (obstacles, power-up blocks, etc.);
- Manoeuvre zones;

A.4 Levels experimented: Amusing and Challeging?

4.1 *Level 1-1: Amusing?*

4.2 *Level 1-1: Challenging?*

4.3 *Level 4-1: Amusing?*

4.4 *Level 4-1: Challenging?*

4.5 *Level 8-2: Amusing?*

4.6 *Level 8-2: Challenging?*

A.5 Do you think difficulty will influence the fun/challenge factor?

- Yes;
- No;

A.6 How many X do you think the level should contain?

6.1 Gaps;

6.2 Hills;

6.3 Enemies;

6.4 Blocks (obstacles, power-up blocks, etc.);

A.7 Survey results

Question	Choices	Answers
A.1	18-23	10
	24-30	5
	31-40	1
	40+	3

Table A.1: Question A.1's results.

Question	Choices	Answers	Question	Choices	Answers
A.2	Straight	2	A.3	Straight	0
	Jumping gaps	4		Jumping gaps	9
	Hills	5		Hills	0
	Enemies	14		Enemies	10
	Blocks	7		Blocks	4
	Manoeuvre zones	9		Manoeuvre zones	14

Table A.2: Questions A.2 and A.3's results.

Question	Choices	Answers	Question	Choices	Answers
4.1	1	0	**4.2**	1	9
	2	0		2	7
	3	5		3	2
	4	9		4	1
	5	5		5	0
4.3	1	1	**4.4**	1	1
	2	1		2	1
	3	3		3	5
	4	6		4	10
	5	8		5	2
4.5	1	0	**4.6**	1	0
	2	4		2	0
	3	4		3	0
	4	4		4	4
	5	7		5	15

Table A.3: Question A.4's results.

Question	Choices	Answers
A.5	Yes	19
	No	0

Table A.4: Question A.5's results.

Question	Choices	Answers	Question	Choices	Answers
6.1	1	0	**6.2**	1	0
	2	4		2	5
	3	10		3	10
	4	4		4	4
	5	1		5	0
6.3	1	0	**6.4**	1	2
	2	0		2	1
	3	8		3	9
	4	9		4	6
	5	2		5	0

Table A.5: Question A.6's results.

Appendix B

Second Survey

B.1 Age group

- 18-23;
- 24-30;
- 31-40;
- 40+;

B.2 Levels experimented: Amusing and Challenging?

4.1 *Easy generated level: Amusing?*

4.2 *Easy generated level: Challenging?*

4.3 *Normal generated level: Amusing?*

4.4 *Normal generated level: Challenging?*

4.5 *Hard generated level: Amusing?*

4.6 *Hard generated level: Challenging?*

B.3 Which zones did you find more entertaining?

- Straight;
- Jumping gaps;
- Hills;
- Enemies;

- Blocks (obstacles, power-up blocks, etc.);
- Manoeuvre zones;

B.4 Which zones did you find more challenging?

- Straight;
- Jumping gaps;
- Hills;
- Enemies;
- Blocks (obstacles, power-up blocks, etc.);
- Manoeuvre zones;

B.5 Comparing the played level to original *Mario* levels, would you say the generated ones would blend in?

- Yes;
- No;

B.6 Survey results

Question	Choices	Answers
B.1	18-23	10
	24-30	4
	31-40	2
	40+	3

Table B.1: Question B.1's results.

Question	Choices	Answers	Question	Choices	Answers
4.1	1	0	**4.2**	1	2
	2	0		2	4
	3	6		3	9
	4	6		4	3
	5	7		5	1
4.3	1	0	**4.4**	1	0
	2	0		2	3
	3	2		3	3
	4	9		4	7
	5	8		5	6
4.5	1	0	**4.6**	1	0
	2	1		2	0
	3	4		3	0
	4	0		4	0
	5	14		5	19

Table B.2: Question B.2's results.

Question	Choices	Answers	Question	Choices	Answers
B.3	Straight	5	B.4	Straight	2
	Jump	6		Jump	3
	Hill	7		Hill	6
	Dunking	12		Dunking	11
	Manoeuvre	8		Manoeuvre	10
	Wall Jump	9		Wall Jump	14

Table B.3: Questions B.3 and B.4's results.

Question	Choices	Answers
B.5	Yes	19
	No	0

Table B.4: Question B.5's results.